Durham E-Theses

Plasma Physics Computations on Emerging Hardware Architectures

JOANNE CLARE CHORLEY

How to cite:

CHORLEY, JOANNE CLARE (2016) Plasma Physics Computations on Emerging Hardware Architectures. Doctoral thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/11912/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

Plasma Physics Computations on Emerging Hardware Architectures

Joanne Chorley

A Thesis presented for the degree of
Doctor of Philosophy

Centre for Advanced Instrumentation
Department of Physics
University of Durham
England

September 2016

Abstract

This thesis explores the potential of emerging hardware architectures to increase the impact of high performance computing in fusion plasma physics research. For next generation tokamaks like ITER, realistic simulations and data-processing tasks will become significantly more demanding of computational resources than current facilities. It is therefore essential to investigate how emerging hardware such as the graphics processing unit (GPU) and field-programmable gate array (FPGA) can provide the required computing power for large data-processing tasks and large scale simulations in plasma physics specific computations.

The use of emerging technology is investigated in three areas relevant to nuclear fusion: *(i)* a GPU is used to process the large amount of raw data produced by the synthetic aperture microwave imaging (SAMI) plasma diagnostic, *(ii)* the use of a GPU to accelerate the solution of the Bateman equations which model the evolution of nuclide number densities when subjected to neutron irradiation in tokamaks, and *(iii)* an FPGA-based dataflow engine is applied to compute massive matrix multiplications, a feature of many computational problems in fusion and more generally in scientific computing. The GPU data processing code for SAMI provides a 60x acceleration over the previous IDL-based code, enabling inter-shot analysis in future campaigns and the data-mining (and therefore analysis) of stored raw data from previous MAST campaigns. The feasibility of porting the whole Bateman solver to a GPU system is demonstrated and verified against the industry standard FISPACT code. Finally a dataflow approach to matrix multiplication is shown to provide a substantial acceleration compared to CPU-based approaches and, whilst not performing as well as a GPU for this particular problem, is shown to be much more energy efficient.

Emerging hardware technologies will no doubt continue to provide a positive contribution in terms of performance to many areas of fusion research and several exciting new developments are on the horizon with tighter integration of GPUs and FPGAs with their host central processor units. This should not only improve performance and reduce data transfer bottlenecks, but also allow more user-friendly programming tools to be developed. All of this has implications for ITER and beyond where emerging hardware technologies will no doubt provide the key to delivering the computing power required to handle the large amounts of data and more realistic simulations demanded by these complex systems.

Declaration

The work in this thesis is based on research carried out at the Centre for Advanced Instrumentation, Department of Physics, England, and at the Culham Centre for Fusion Energy, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Some of the original work presented in this thesis has been published in the following journal article:

“GPU-based data processing for 2-D microwave imaging on MAST”, J.C. Chorley et al., *Fusion Science and Technology*, **69**, 643 (2016)

Acknowledgements

I would like to thank my supervisors Ray Sharples and Nigel Dipper for their continued support and advice throughout. Their guidance has been of great value. I would also like to thank Roddy Vann for his proposal for the work in Chapter 3, and for his advice and supervison when carrying out the work. Additionally I'd like to thank Lee Morgan and Michael Fleming for their guidance in Chapter 4. Finally I would like to thank Rob Akers for supporting my move to Culham and helping me get settled there.

I would like to thank the Fusion Doctoral Training Network/Fusion Centre for Doctoral Training for providing me with an excellent start to my PhD and for providing a brilliant network of students and academics.

Finally, I would like to thank my friends, family and especially Edd for being my motivation and providing encouragement when I needed it most.

The work in this thesis is funded by EPSRC grant EP/K504178 for the Fusion Doctoral Training Network. The work in Chapter 3 is funded by the SAMI grant EP/H016732, the UK Magnetic Fusion Research Programme EP/I501045 and the EUROfusion pedestal grant ER-WP15 CCFE-03.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Experiments, theory and computational physics are all equally important tools to assist scientific research. Theoretical models can be validated with computational modelling and experimental observations. As computer performance has improved with Moore's Law [1], the size, accuracy and detail of the problem being investigated has improved. The exponential growth in computer power means problems that were thought impossible 10-20 years ago can now be solved. Many interesting and complex plasma physics problems can be studied with computational methods. Additionally, improvements in algorithmic efficiency have increased performance further and now simulations are an important tool in fusion reactor performance predictions and design [2]. This chapter discusses the development of computational plasma physics to present day, inline with the developments in computing technology, high-lighting the necessity to consider new emerging architectures to maintain the improvement rate of simulation and data processing capabilities.

1.1 Development of computational plasma physics

1.1.1 Early history

Plasma physics study began in ernest in the 1950s and 1960s with interest in the applications to nuclear fusion for energy and for weapons development. At the same time, high-speed electronic computers were being developed. These early computers, whilst impressive, were quite restricted in terms of the plasma physics problems of interest they could simulate; for example, the effect of small angle collisions on longitudinal plasma oscillations was simulated in 1958 [3] and run on the IBM 704, one of the first machines with dedicated floating-point arithmetic hardware units. The Princeton Plasma Physics Laboratory (PPPL) aquired the IBM 650 in 1959 which had a memory of 8000 words and a speed which limited the models run on it to be one-dimensional containing only a couple of hundred particles [4]. Nevertheless these limitations did not affect the ability of the simulation to accurately model the collective behaviour of plasma.

The pioneers of early plasma physics computer simulations were Dawson and Buneman in the 1960s. Dawson used sheet charges on the high-speed computers of the time [5] whereas Buneman followed the trajectories of charged particles. Both used small systems of a few hundred to a thousand particles or sheet charges limited by the current technology available; for example, Buneman's simulations ran on a Univac 1103 AF digital machine, taking 2 hours [6]. Hockney's simulations a little later [7] [8], performed on a 100 kilo-floating-point operations per second (kFLOPS) IBM 7090 with a few thousand particles, paved the way for large scale computational plasma simulations. As technology improved in terms of speed and memory size, larger simulations of fifty thousand particles could be traced on machines such as the IBM 360 and CDC 6600 [9].

By the 1970s, plasma simulation employing particle models was fairly well established and making contributions to plasma theory and the interpretation of experiments. As the computational performance improved, two-dimensional simulations involving $10^4 - 10^5$ particles were more commonly performed [10]. However, the high runtime of these simulations restricted the number of parameters that could be studied. Due to the computational power required, three-dimensional simulations were much rarer. Computational plasma physicists at the time knew that due to Moore's law speed and memory capacity would increase by a factor of 10-100 in a few years, making plasma simulations much more practical in the near future [10].

Whilst the complexity of computational plasma physics required problems to be carried out on the fastest supercomputers, the reliability of machines such as the CDC 7600 at Lawrence Livermore National Laboratory was a problem. The notorious CDC 7600 computer crashes limited how far a simulation could be run and still produce recoverable data [11]. Many university groups used the CDC 7600; for example the Stanford group ran SPLASH, a 3-dimensional fully electromagnetic particle code with one to two hundred thousand particles [12]. However problematic the CDC 7600 was, the machine was a key step towards modern day computers and many of its design components are present in machines today.

Development in parallel computing in the 1980s and 1990s led to advancement in the Numerical Tokamak project [13]. The modern parallel computers of the time, for example the four vector processor CRAY-2, could model tokamaks and turbulent transport using gyro-kinetic particle models with 8 million particles whose motion was followed in three-dimensions [14]. The speed and memories of serial computers of the same time were three orders of magnitude smaller than required for the simulation [4], hence the need to move to parallel computers. Large scale plasma modeling has been made possible by the increase in computing power. From the late 1990s, simulations could handle the complexity of many real experiments [15]. Fully three-dimensional simulations consisting of an increasing number of particles were able to be performed; for example, over 100 million particles were able to run on the IBM SP2 machine [16]. Moving to multiple instruction multiple data (MIMD) computers like the 512 node Intel Touchstone Delta, 3D electromagnetic plasma particle simulations could be run 58 times faster than on a single CPU Cray C90 or 116 times faster than a Cray Y-MP [17]. Admittedly, the code was not optimized to run on the C90 and only compiled with the Cray automatic vectorization and optimization so real Cray performance would be somewhat improved. Nevertheless the performance gain from moving to a many-core machine is evident. Later, from 2000 onwards, global gyrokinetic codes were implemented on massively parallel platforms like the CRAY T3E [18], which was the first machine to break the teraflop barrier, achieving this in 1998. The Cray T3E was a distributed memory machine requiring Message Passing Interface (MPI) to pass particle and grid information between processors.

1.1.2 Advances in physics studies made possible by advances in computational capabilities

The simulations of Dawson in the early 1960s [5] were based on a simple one-dimensional model of a plasma using sheet charges. Whilst this model was simple, it was able to reproduce a number of real plasma properties and model the plasma quite accurately. As the model is simple and small, statistical mechanics may be applied and statistical theories obtained for the behaviour of the system. Thus the simulated system properties can be compared to the theoretical system properties, and at that time, this could be done in more complete detail than with a real experimental plasma. Dawson was able to show good agreement between theory and simulation for multiple plasma properties: the velocity distribution, Debye shielding, drag, diffusion in velocity space, Landau damping, amplitude distribution function and the electric field distribution. In the early days computational plasma physics was used to validate existing theory and as a guide for non-equilibrium properties and non-linear phenomena.

Advances in computational technology meant by the early 1970s complicated non-linear and turbulent phenomena in plasma could be investigated [10]. Simulations illuminated the mechanisms involved which in turn allowed theories for the observed effects to be postulated. In a short space of time, simulation capability had improved such that it could inform new theoretical theory for plasma processes. For example, computer simulation gave the first quantitative estimate of instabilities associated with high-frequency oscillating electric fields and the associated non-linear absorption of radiation. Whilst experiment was not precise enough to validate the simulation, energetic particles produced by turbulence were present and there was good qualitative agreement between experiment and simulation for the particle energy.

Another example arises from the simulation of plasma diffusion across a magnetic field. Two-dimensional simulations could be done, but often different simplifications to a model led to different results. One two-dimensional study [19] using a parallel uniform magnetic field and particles with the $\mathbf{E} \times \mathbf{B}$ drift velocity showed the plasma diffusion across the magnetic field was Bohm diffusion. However, another two-dimensional study from around the same time keeping the full dynamics of the particles showed the diffusion was classical collisional diffusion. Once the computational technology was advanced enough, this lead to a fully three-dimensional simulation being carried out [10] which verified the classical collisional theory of plasma diffusion across a magnetic field was unsuitable at large magnetic fields. At large magnetic fields, collective modes give rise to convective motion in the plasma which dominates the transport. This effect was later observed in experiment.

Plasma physics made huge strides in the 80s and 90s by being able to predict the results of experimental observations and by explaining natural plasma phenomena. These advances are due to the simultaneous advances in plasma diagnostics, in plasma theory, and particularly through vastly improved computer modelling capability [15]. This improved computational power provded the capability for gyrokinetic simulation of tokamaks to study turbulence. Until the mid 1990s, turbulent transport calculations had demanded more computing power than was available [13] but for the first time, with the present day (1995) computing technology, calculations in a true toroidal geometry that directly simulate a small to moderate-sized tokamak or a slice of a large tokamak could be performed. The advances in computational capability meant three-dimensional toroidal and quasi-toroidal simulations of ion-temperature gradient modes and trapped-particle instabilities relevant to core transport could be performed. Additionally, hybrid gyrokinetic-magnetohydrodynamic three-dimensional simulations of energetic particles to study saw-tooth activity, fishbone oscillations, and toroidal Alfven eigenmodes excited by fusion alphas were able to be performed [20].

It is clear advances in computational technology led to advancement in plasma physics studies. Initially, simulation was mainly used to verify existing plasma theory. As computational power improved, and thus simulation capabilities increased, new plasma effects were observed which in turn led to the development of new theories, or the inclusion of new physics in existing theories. From simulating small scale one-dimensional plasmas, the advancement in computational technology now provides the capability to perform full tokamak simulations in the toroidal geometry.

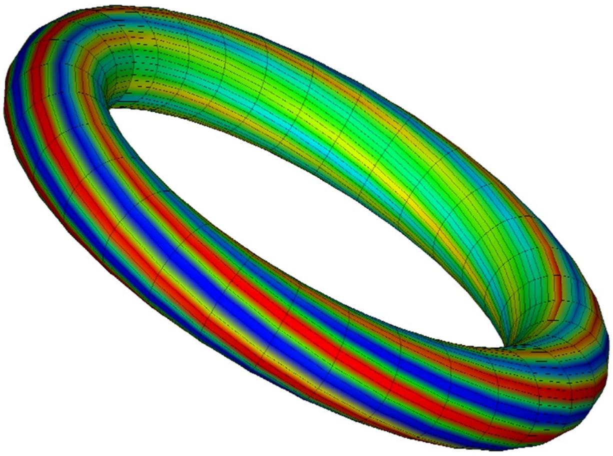


Figure 1.1: GTC toroidal flux surface showing the quasi-2D structure of the electrostatic potential generated by the plasma particles. The elongated structures follow the magnetic field lines [26].

1.1.3 Modern computational plasma physics

The field of computational plasma physics has grown to encompass many areas of fusion. There are many codes studying a wide range of problems and phenomena, and many codes are multi-purpose. What follows is a brief review of the literature for some popular codes that are currently used, but in no way is meant as a comprehensive evaluation of this vast field.

Gyrokinetic codes

Gyrokinetic turbulence codes can be separated into those based on Lagrangian-PIC methods such as GTC (Gyrokinetic Toroidal Code) [21] and those based on Eulerian schemes such as GS2 [22] [23], GENE (Gyrokinetic Electromagnetic Numerical Experiment) [24] and GYRO [25]. GTC was developed in 1998 and designed to run on massively parallel machines like the CRAY-T3E. GTC was one of the first codes to run in hybrid mode MPI-OpenMP (Open Multi-Processing) on an IBM SP Power 3, allowing an increase in concurrency from 64-processor simulations to the full use of 1,024 processors [26], modelling one billion particles. Later, GTC needed to be MPI-only in order to run effectively on a large number of processors of the Earth Simulator, achieving 3.7 Teraflops on 2048 cores. The GTC torus geometry

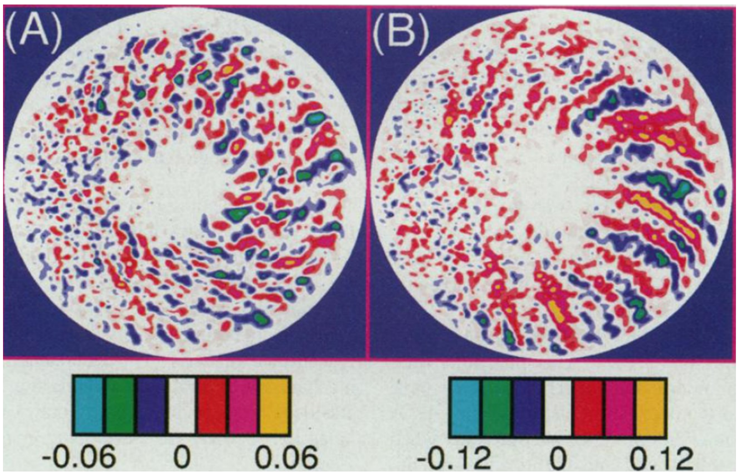


Figure 1.2: GTC poloidal contour plots of fluctuation potential ($e\phi/T_i$) in the steady state of non-linear global simulation with E × B flows included (A) and with the flows suppressed (B) [21].

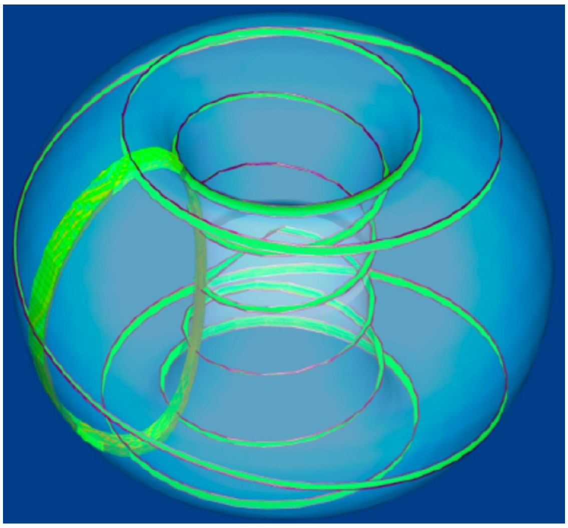


Figure 1.3: The green patches denote the reduced space time domain used in TRINITY, each of which corresponds to a nonlinear gyrokinetic flux tube simulation, and are the grid points in the coarse space-time mesh used to solve the transport equations [28].

is shown in Fig. 1.1 [26] and the poloidal contour plots of the fluctuation potential in Fig. 1.2 [21]. The local non-linear gyrokinetic code GS2 was first developed in the late 1990's and together with GENE (2011) is used to calculate the turbulent fluxes to be fed to the multiscale gyrokinetic transport code TRINITY [27] developed in 2008. TRINITY couples transport and gyrokinetic turbulence and can investigate micro-turbulence. Some simulations using this code are discussed [28] and indicate the long runtime of the code; an L-mode discharge from Joint European Torus (JET) (shot #19649) taking four hours on 5760 CRAY XT4 processors for 15 time steps, H-mode discharges from JET (shot #42982) taking just under ten hours on 8640 CRAY XT4 processors with 20 time steps and ASDEX Upgrade (shot #13151) taking 24 hours for 16 transport time steps on 16384 CRAY XT4 processors. The space-time mesh used to solve the transport equations is shown in Fig. 1.3 [28]. These codes are still widely used as the industry standard today but have had to be adapted for larger machines as more and more parallel processors are added. This is not always beneficial as increasing the number of processors adds extra communication costs and so some codes have needed to be redesigned to exploit the full parallelism of the machine. Even so, many reports show that there is a limit to increasing performance by adding more processors. In 2005 the GTC code shows strong scaling (fixed problem size on an increasing number of processors) on a number of machines, up to 2048 cores shown in Fig. 1.4 [26], whereas the GYRO code's performance [29] doesn't scale as well for fewer processors (see Fig. 1.5). A few years later, the performance of the GENE code on more than 10000 cores was investigated and is presented in Fig. 1.6 [24]. Again, the scaling weakens for higher core count, in this case for a much larger number of cores, past 10000 cores. More recently still, in 2015 the GS2 code was optimized to improve performance [30], which was achieved, but the runtime does not significantly improve past a certain number of cores as seen in Fig. 1.7.

Magneto-hydrodynamic codes

Another area of computational plasma physics is magneto-hydrodynamic (MHD) codes, such as BOUT [31], ELITE [32] [33], JOREK [34] and BOUT++ [35], the lat-

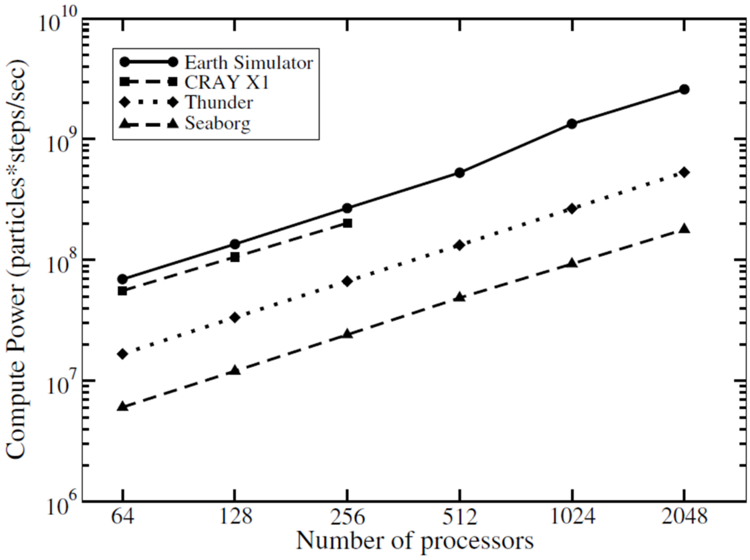


Figure 1.4: GTC shows strong scaling (keeping the problem size fixed and increasing the number of processors used) on a number of machines up to 2048 cores [26].

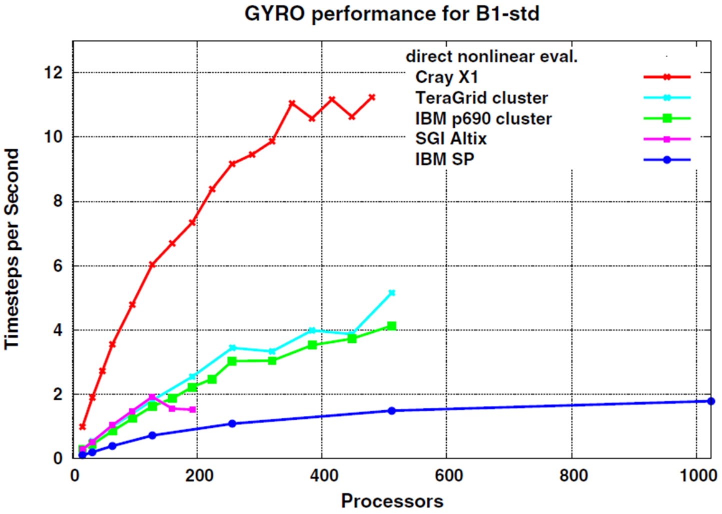


Figure 1.5: The performance of the GYRO code on different machines [29]. It appears that adding extra processors does not increase performance past a certain point, most notably on the IBM SP.

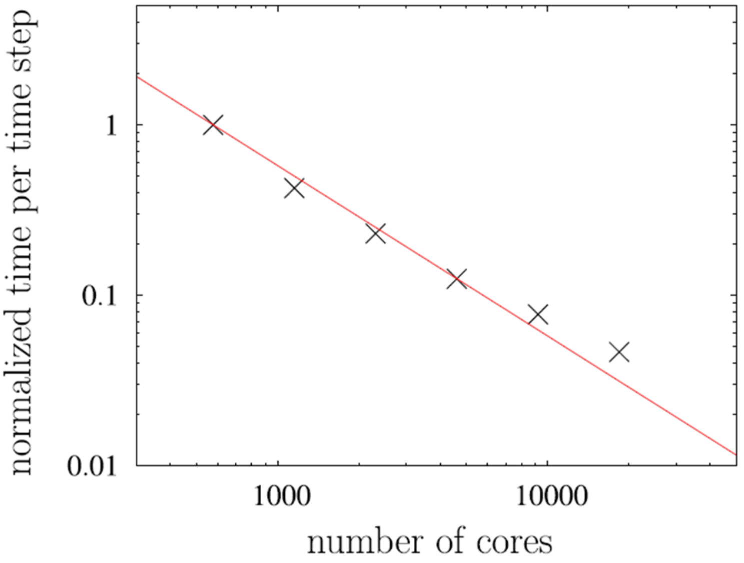


Figure 1.6: Strong scaling is observed for GENE on Hector [24], but as the number of cores exceeds 10000, the scaling weakens. The red line shows a linear scaling with the number of cores.

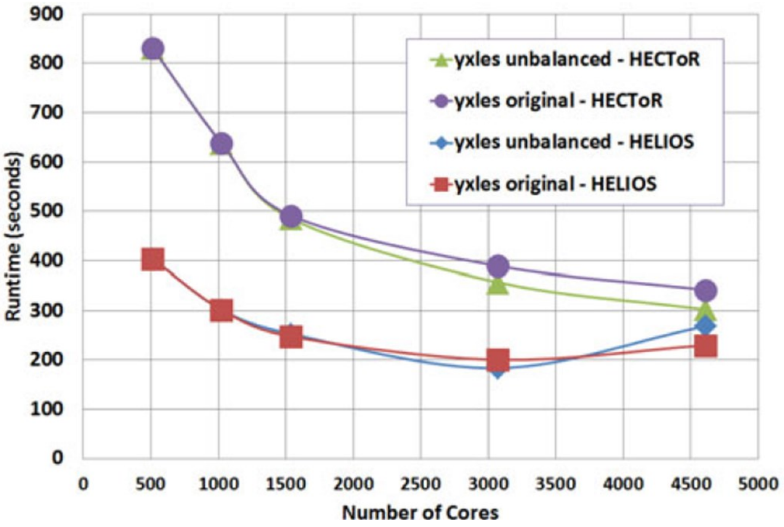


Figure 1.7: Comparison of the performance of the original and newly optimised GS2 decompositions on Hector and Helios [30] Good scaling is observed up to 1536 processors but as the processor count is increased further the scaling gets worse.

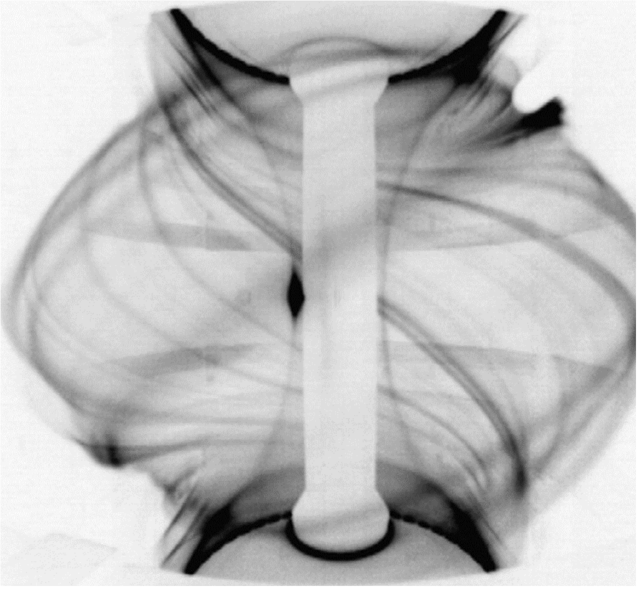


Figure 1.8: An image of an ELM event in the MAST tokamak showing eruption of filaments from the plasma edge. ELMs are modelled by the BOUT++ code [35].

ter developed from BOUT and performing parallel plasma fluid simulations. These codes are used to aid the understanding of edge phenomena such as edge localised modes (ELMs) which is critical to the realisation of the International Thermonuclear Experimental Reactor (ITER) [36]. An ELM event in Mega Amp Spherical Tokamak (MAST) [37] is shown in Fig. 1.8. The initial BOUT simulations were carried out on the CRAY T3E at National Energy Research Scientific Computing Center, and the IBM SP machine, modelling, and benchmarked against, the HT-7 tokamak. JOREK is a non-linear extended MHD code, parallelised using MPI and OpenMP.

BOUT++ simulates 3D fluid equations in curvilinear coordinates, performing large-scale $10^7 - 10^8$ variable simulation, and was the first code to successfully perform initial-value simulations with realistic resistivity of ELMs [35]. ELMs are also modelled with the JOREK, NIMROD and M3D-c1 codes (amongst others). BOUT++ scales well for increasing number of processors for a fixed problem size (strong scaling) as can be seen from Fig. 1.9. This was performed using the National Energy Research Scientific Computing Franklin machine. The efficiency is given as:

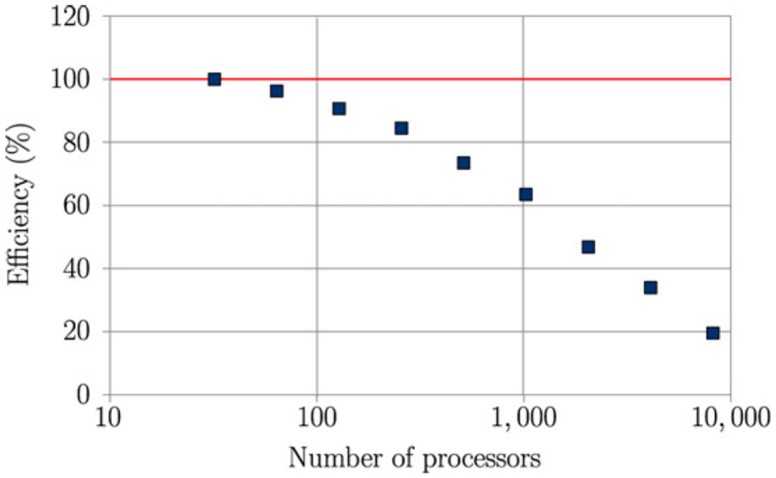


Figure 1.9: Efficient scaling of BOUT++ for a fixed problem size on up to 8192 processors. [35]

$$\epsilon(N_p) = 100\frac{N_{p0}T(N_{p0})}{N_pT(N_p)} \tag{1.1.1}$$

Fig. 1.9 shows that for a fixed problem size, 2048 processors can be used at approximately 50% efficiency relative to 32 processors. Therefore, by rearranging equation 1.1.1, $T(N_{32}) = 32T(N_{2048})$ and the time taken to run a fixed problem size on 32 processors is 32 times longer than the time taken to run on 2048 processors. Similarly, 8192 processors can be used at 20% efficiency relative to 32 processors so $T(N_{32}) = 51.2T(N_{8192})$ and running on 8192 processors is 51 times faster than running on 32 processors. Therefore, for a fixed problem size, increasing the number of processors decreases the runtime of a BOUT++ simulation.

More recently, weak scaling (keeping the amount of work each processor does fixed but increasing the overall problem size with the number of processors) has been performed on BOUT++ (see Fig. 1.10) which shows whilst there is good scaling for the size of the problem, m, the scaling with processor number is poor for large number of processors due to global gather and scatter operations [38]. This communication cost for increasing number of processors is commonly seen in many MPI-based applications.

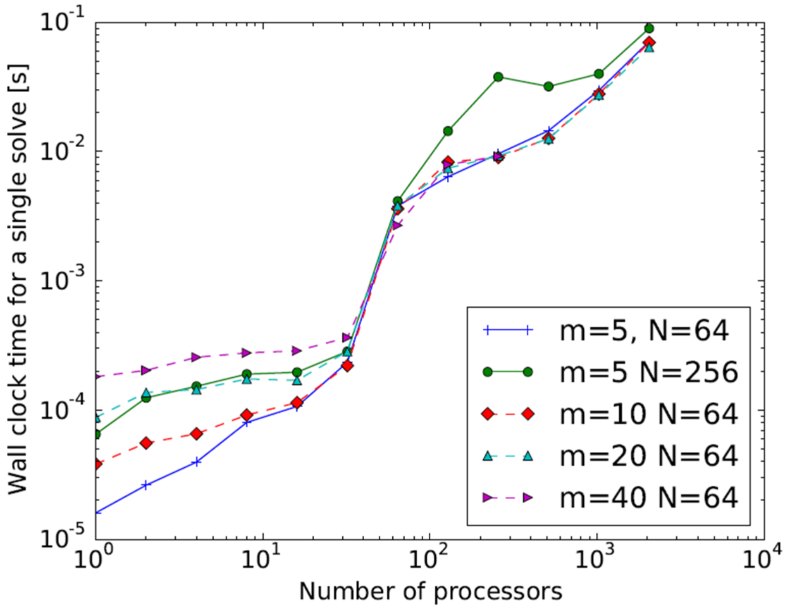


Figure 1.10: BOUT++ simulations on Hector with 32 cores per node and up to a maximum of 2048 cores. m is the size of each system, and N is the number of separate systems which are inverted simultaneously [38]. Past 32 cores, the inter-node communication dominates.

1.1.4 The future of computational plasma physics

Moore's Law and frequency scaling are coming to an end. Additionally, strong scaling with core count does not exist for many codes when running on a large number of cores. Adding extra cores does not necessarily improve the performance in a number of simulations as communication costs can start to dominate as seen above. Alternative highly parallel hardware architectures like the Graphics Processing Unit (GPU), Intel Many-Integrated-Core (MIC) and Field-Programmable Gate-Array (FPGA) based Data-Flow Engine (DFE) have been developed. These architectures can provide large acceleration of an application but often require a redesign of the algorithm that specifically targets the hardware. Many of the above types of codes are being redesigned for the GPU; for example GTC has been redesigned for the GPU [39], as has GENE [40], and PSC is another particle-in-cell (PIC) code redesigned for the GPU [41]. PIC codes follow the trajectories of a large number of charged particles in self-consistent electromagnetic (or electrostatic) fields [42]. In the PIC scheme, particles are defined in both position and velocity space where as fields are only defined at discrete locations in space at mesh points [43]. Both fields and particles are also defined at discrete times. The algorithm works by first advancing the particle and field values in time. Next the particle equations of motion are advanced one time step, using fields interpolated from the discrete grid to the continuous particle locations. The fields are then advanced one time step, the particle positions updated and the time step loop repeats.

The performance benefits GPU technology provides for these applications is evident as PIC is demanding in terms of computing power, since a large number of particles are needed to represent the underlying physics [44]. PIC codes are good candidates for being ported to the GPU architecture as the large number of particles can be followed simultaneously for each time step on the GPU, rather than sequentially on non-parallel architectures. It is important to consider the type of problem to be accelerated, before choosing a technology to do so as some types of problems are more suited to one type of technology. For example single instruction multiple data (SIMD) problems are well suited to a GPU architecture [45]. Other problems such as iterative methods are generally not very well suited to the GPU architecture, requiring more thought to achieve an acceleration [46]. Iterative methods are more suited to a different technology such as FPGAs [47]. A lot of PIC codes are being ported to GPUs [48], where as finite difference based codes might be better suited to a dataflow architecture as has been demonstrated with the finite difference dataflow application MaxGenFD [49]. Additionally, these technologies can be used for compute intensive tasks other than simulation, such as data-processing, data-reduction and signal-processing.

1.2 Emerging architectures in computational physics and plasma physics

1.2.1 Examples of projects using emerging architectures

As performance improvements have slowed down and are not as easily achieved, emerging architectures such as the GPU and FPGA have become more popular

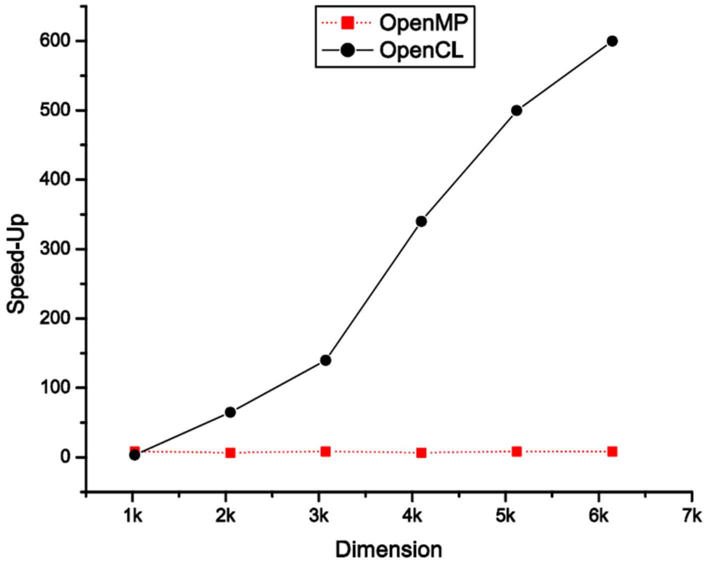


Figure 1.11: Speed-up comparison of dense matrix multiplication for an OpenMP and OpenCL GPU implementation compared to sequential implementation. The OpenCL implementation running on the GPU performs extremely well for large matrix dimension [51].

in computational physics. Alternative technologies such as the GPU have become very popular for some highly computationally demanding tasks due to the increased floating-point compute power, greater memory bandwidth and better energy efficiency they provide compared to modern CPUs [50]. For example, multiplying two large, dense matrices on both a multi-core architecture using only Open Multi-Processing (OpenMP) and on a GPU shows that for larger problem sizes the GPU far outperforms the multi-core OpenMP implementation as seen in Fig. 1.11 [51]. Solutions involving only OpenMP or only Message Passing Interface (MPI) do not perform as well as GPU implementations [52]. Increasingly the field of computational science is turning more to accelerators like the GPU, field-programmable gate arrays (FPGAs) and Intel Many-Integrated Core (MIC) to achieve the computational power required as we approach the exascale [53]. Many problems in computational science such as n-body problems, collision detection, probabilistic Potts model simulations and Cellular Automata simulations, are data parallel and ideal for acceleration with a GPU [54]. The larger problem can be broken down into many independent smaller problems which can be solved simultaneously with a single instruction operating on multiple data (SIMD). Iterative solvers where the next iteration depends on results from the previous iteration do not follow the SIMD model and as such are poor candidates for acceleration with a GPU.

Large data experiments such as the Square Kilometre Array (SKA) are looking towards alternative technologies and specialised hardware such as the GPU, in order to ease the big data problem. Beam forming in radio astronomy has higher performance and is more energy efficient on a GPU system compared to a multi-core CPU system [55] and a recent analysis [56] concluded that for SKA, novel hardware and system architectures need to be developed to achieve the required power efficiency and compute capabilities. The European Organization for Nuclear Research (CERN) and Large Hadron Collider (LHC) have investigated the potential of using GPUs in the high-level trigger algorithms used to select the interesting data to reduce the raw data obtained from this experiment at the nominal LHC collision rate of 40MHz or every 25 nanoseconds [57].

FPGAs will also be important for the SKA which needs to process signals from multiple antennas [58] and FPGAs have demonstrated their importance in the trigger and data acquisition system of the ATLAS particle detector experiment at the LHC with the discovery of the Higgs boson [59]. Recently, the Maxeler dataflow engine [60] has gained popularity in the field of HPC on FPGAs. Problems are reformatted as a continuous flow of data into the dataflow engine and inputs are buffered to maintain a constant but high bandwidth data stream. However the field of high performance computing on FPGAs is in its infancy due to a lack of developed compilers and a complex development cycle [61] and it is accelerators like the GPU and MIC that are currently the most popular in computational science. Indeed, at the time of writing, three of the top ten supercomputers in the Top500 list are hybrid machines [62] utilizing either GPUs or MICs as an accelerator or coprocessor; it is clear this is the direction in which computational science is going. Often on these machines, it is a heterogeneous programming model featuring GPU, MPI and OpenMP that achieves the best performance [63].

1.2.2 Examples of emerging technology used in plasma physics

For current and certainly next generation fusion devices such as ITER, specialised hardware and techniques need to be employed in data processing tasks as the amount of data produced in experiments becomes large and difficult to handle with traditional approaches [64]. We are entering a mode of operation for tokamaks in which large amounts of data are produced. For tokamaks such as NSTX, in 2004 camera data amounted to 300 MB/pulse, or 600 MB/pulse if all the data had been archived whereas shortly afterwards in 2006, one camera alone can acquire 2GB/pulse [65]. For next generation tokamaks such as ITER, between 160GB/pulse, extrapolating from existing devices, to 100TB/pulse if physics reasoning is considered and predicted acquisition rates will be produced [66]. It is therefore worthwhile to investigate the suitability of emerging hardware like the GPU for massive data processing tasks such as those in real-time fast control systems [67]. What follows are a few examples of the use of GPUs and FPGAs within the fusion community, meant to give an illustration of areas where these technologies have been successfully utilized, but again, it is not an exhaustive review of the field.

GPUs in fusion

GPUs have been used fairly extensively in areas across fusion, for simulation, computation and control systems. Examples in simulation and computation, in addition to the ported PIC codes discussed earlier, include the Monte-Carlo code LOCUST-GPU [68] which simulates millions of fast ions in a few hours, and the computation of magnetic field maps using the MISTIC code [69] giving a speed-up of at least 16 over the CPU-based approach (Fig. 1.12). For problems such as EFIT, GPUs have been demonstrated to be well suited to the problem with P-EFIT [70]. On the use of GPUs in control systems for tokamaks, much work has been done. The plasma control system for the HBT-EP tokamak successfully uses GPUs to magnetically control the 3D perturbed equilibrium state [71], as does the self adaptive sampling rate data acquisition system installed on JET [72]. The performance of a fast plant system control prototype for ITER using GPUs to do plasma preprocessing is compared to a state-of-the-art CPU based system [64], with the GPU system performing

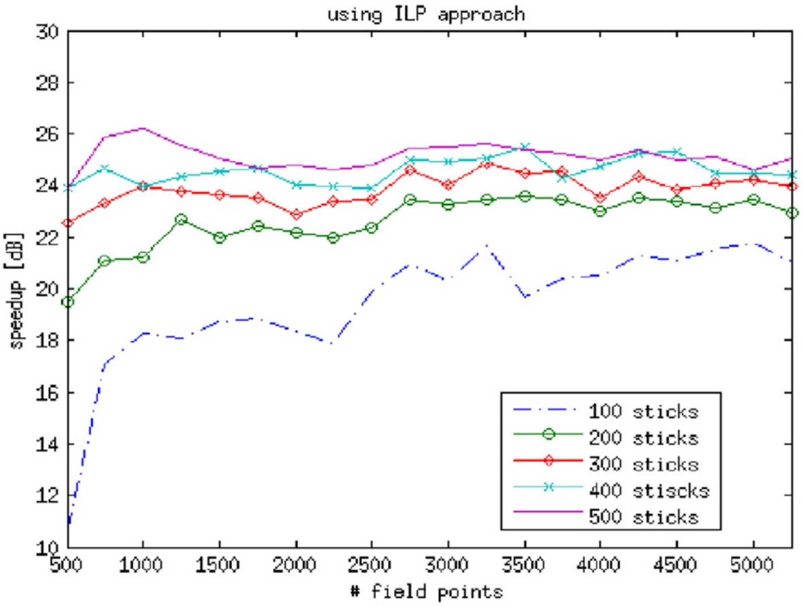


Figure 1.12: Speed-up of MISTIC code on a GPU compared to a CPU where $\text{Speedup}_{dB} = 20 * log_{10}(t_{CPU}/t_{GPU})$ [69]. The sticks discretize a conductor in an active coil, e.g. Central Solenoid coil.

much faster than the CPU system (Fig. 1.13).

FPGAs in fusion

Currently FPGAs are more commonly utilized in signal processing and data acquisition, rather than high performance computing applications. The data-acquisition system for the Large Helical Device is FPGA-based and set a record in the 2005-2006 campaign for acquiring 90GB for one shot, see Fig. 1.14, although normally 3GB per shot is acquired with 80MB/s real-time data acquisition [73]. The TJ-II stellarator diagnostic measures the line averaged electron density in real time which is used for control purposes [74]. Another real time FPGA system is used for disruption detection in JET [75] which uses machine learning techniques based on support vector machines (SVMs) for disruption prediction. The FPGA SVM system has been validated with JPS (which is the real time alarm system for incoming disruptions on JET) as shown in Fig. 1.15 [76]. The FPGA SVM predictor detects more disruptions for both unintentional disruptions (those that happen in normal tokamak operation) and intentional disruptions (those that are triggered) in the first 200ms

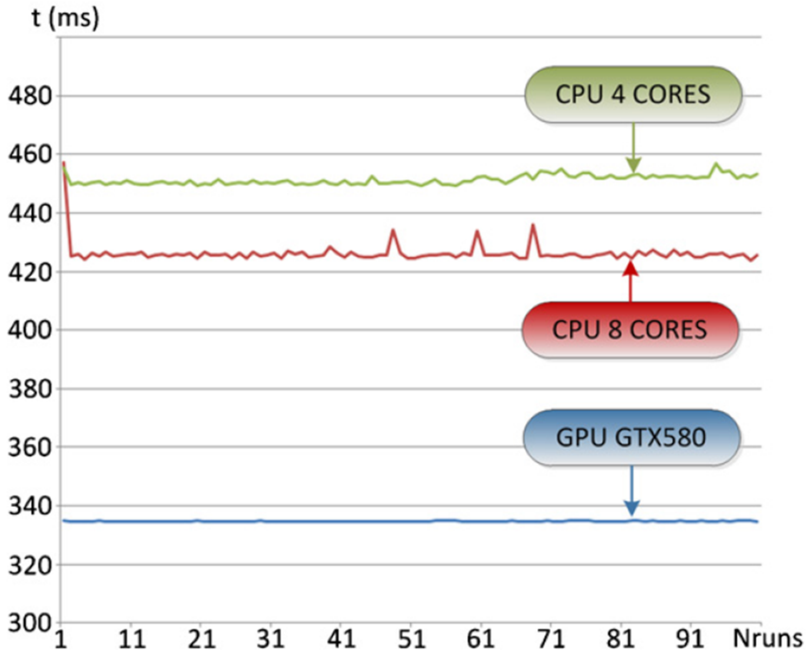


Figure 1.13: GPU execution times compared to CPU execution times for 4 and 8 cores [64].

which is coherent with the training of the system.

FPGAs have also been used for fast filtering in logic such as for the MAST Thomson scattering system [77] and in the neutron diagnostic data acquisition system at JET to perform real-time data validation [78]. FPGA technology has been used in image processing tasks such as that for the real-time infra-red image acquisition and processing system at Tore Supra which is vital to avoid damage to the plasma facing wall components of the tokamak [79].

1.2.3 Scope of thesis

This introduction has highlighted the close relationship between the capability of computation in plasma physics and the developments in computational technology. As a new era of exascale computing is beginning, it is becoming more common for alternative, accelerating technology to be used in a wide range of fields, including the fusion industry. Nevertheless GPU programming for fusion applications is still relatively rare, and FPGA programming for general high performance computing

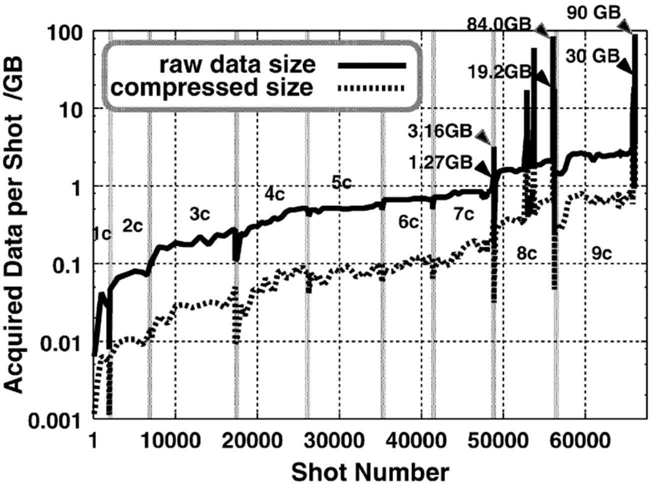
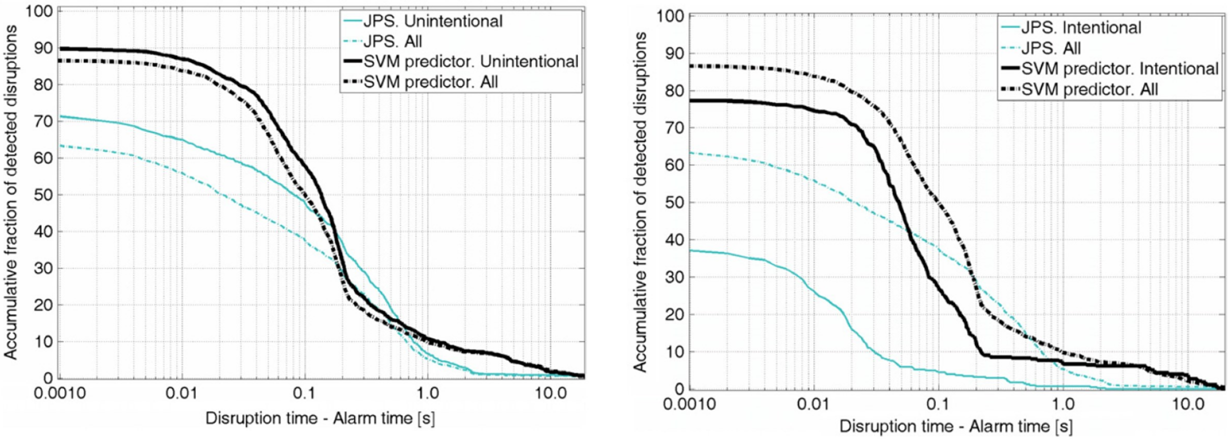


Figure 1.14: Acquired data per shot for the LABCOM data acquisition system for LHD. A new world record of 90 GB per shot was set in 2005 - 2006 campaign [73].



(a)

(b)

Figure 1.15: Warning times (disruption time minus alarm time) for all the disruptive pulses from JET campaign C1 till campaign C19; (a) the cumulative percentages of unintentional and all detected disruptions for the FPGA SVM method are compared with the JET JPS alarm results, (b) the cumulative percentages of intentional and all detected disruptions for the FPGA SVM method are compared with the JET JPS alarm results [76].

applications is a field very much in its infancy. In this thesis, the use of these emerging technologies is further investigated for some specific new fusion applications. GPUs are used in two ways to accelerate two different fusion applications, one online and one off-line. In the first, GPUs are used to accelerate the data-processing of the high data-rate SAMI diagnostic [80] expanding on their use in other large data experiments such as CERN. In the second application the solution of the Bateman equations which are central to the FISPACT-II nuclear burn-up simulations are ported to GPUs, again following other plasma physics simulations that have started to be ported to GPUs. Finally, the potential of an FPGA dataflow engine is investigated. A massive dense matrix multiply problem, which is common to many fusion applications, was chosen to accelerate and to demonstrate the capabilities of this technology.

The thesis is organised as follows. Chapter 2 introduces the emerging architectures, the GPU and FPGA (in the form of a dataflow engine), which have been used to carry out the work. The hardware architecture of each technology and the software in each of the programming models is described. Chapter 3 and Chapter 4 discuss the use of the GPU in the high data rate SAMI diagnostic data processing task and the neutronics Bateman solver calculations. Chapter 5 investigates a fusion relevant problem on the dataflow engine. In Chapter 6 a summary is presented, highlighting the benefits these new and cutting edge technologies provide to many problems and challenges encountered across fusion research and their importance going forward for projects such as ITER and DEMOnstration Power Plant (DEMO) [81].

Chapter 2

Emerging architectures

Conventional serial computing has only a single CPU so there is a logical sequence of operations within a program. The CPU executes the instructions in order, with only one operation in action at one time. Parallel computing uses many CPUs to produce the same result in less time, or to handle larger problem sizes. A popular method for programming shared-memory machines is to use a conventional language with additional compiler directives such as OpenMP which is discussed in detail in [82]. Programming with a distributed memory architecture uses MPI. It follows the SPMD paradigm (Single Program Multiple Data) so the code may branch so that different processors do different jobs or work on different copies of the data. A complete description of parallel programming with MPI and OpenMP is given in [83]. High-performance computing applications are now demanding more than processors alone can deliver. In this chapter, a review of three different emerging architectures and accelerating technologies used in high performance computing is given; the graphics processing unit (GPU), the field-programmable gate-array (FPGA) and many-integrated cores (MIC). The different hardware architectures and programming models are discussed and comparisons of benchmark performance for each technology are presented.

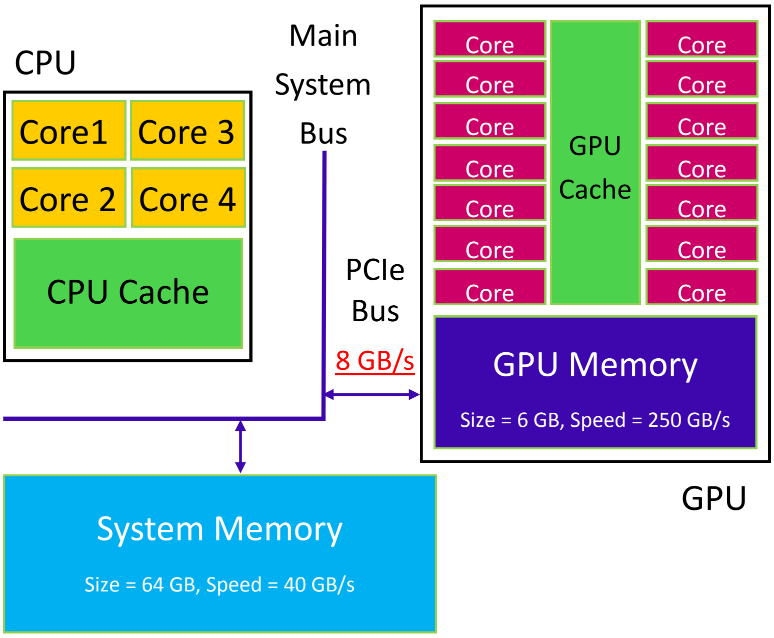


Figure 2.1: A schematic of a hybrid CPU-GPU system adapted from [85]. Memory sizes are typical values and the speed is the maximum memory bandwidth. Different GPUs have different memory size and bandwidth.

2.1 GPUs

2.1.1 GPU architecture

A description of the background, hardware and programming model for GPU computing is given in [84]. A GPU is a massive vector processor meaning the hardware operates on multiple data elements (making up a vector) simultaneously, rather than on only a single data element. GPUs contain hundreds of processing units which can collaborate and have large memory bandwidth. This means that certain types of problems may be solved more efficiently on a GPU. The GPU devotes more transistors to data-processing and is good at doing lots of numerical calculations simultaneously. It is useful as a co-processor as the GPU can be working at the same time as the CPU; GPU efficient calculations are offloaded to the GPU whilst the CPU continues with the rest. Examples of the types of problem well suited to GPUs, including some fusion applications, has been discussed in Chapter 1.

A GPU is attached to a CPU via PCIe (see Fig. 2.1) and expensive parts of

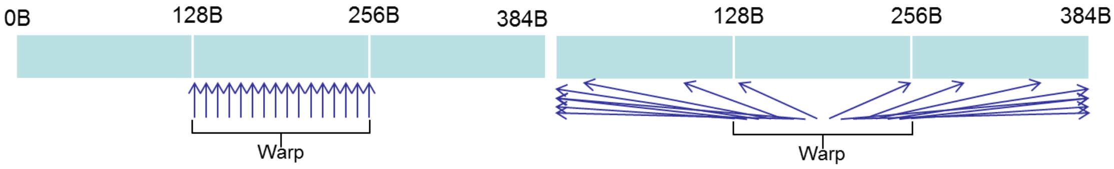


Figure 2.2: A full vector unit of length 32 makes up a warp. One warp is processed simultaneously by the hardware.

a computation are offloaded to the GPU. GPUs work by making massive use of long vector units. A full vector unit, of length 32, is known as a warp, and a warp is processed in parallel by the hardware as shown Fig. 2.2. CUDA threads are a software construct described in section 2.1.2. There are 1536 or 2048 CUDA threads (depending on the GPU) per streaming multi-processor and each CUDA thread is mapped to one element of a hardware vector unit. It is beneficial to have consecutive CUDA threads, making up a warp, point to consecutive data elements in memory.

GPUs are essentially designed to do massive parallel computations and provide the largest acceleration for data parallel problems with all threads executing the same instructions on different data.

2.1.2 GPU software - CUDA parallel programming model

Since 2006, with the introduction of the GeForce 8800, GPUs have been more readily adopted by the scientific community for high performance computing. This is mostly due to the effort made by companies like Nvidia in the development of languages and programming paradigms such as CUDA which make programming GPUs more accessible to the scientific community [86]. Prior to this, it was difficult to write programs to carry out scientific computation on a GPU because non-graphics computations needed to be expressed with a graphics API such as OpenGL [87]. On the software level, CUDA threads are grouped into thread blocks, with a maximum hardware limit of 1024 threads per block so that a thread block can fit onto one streaming multi-processor. Thread blocks are arranged in a grid which is then used to call the CUDA kernel so there is a fixed number of CUDA threads in flight throughout the kernel execution. A small example is illustrated in Fig. 2.3 for a

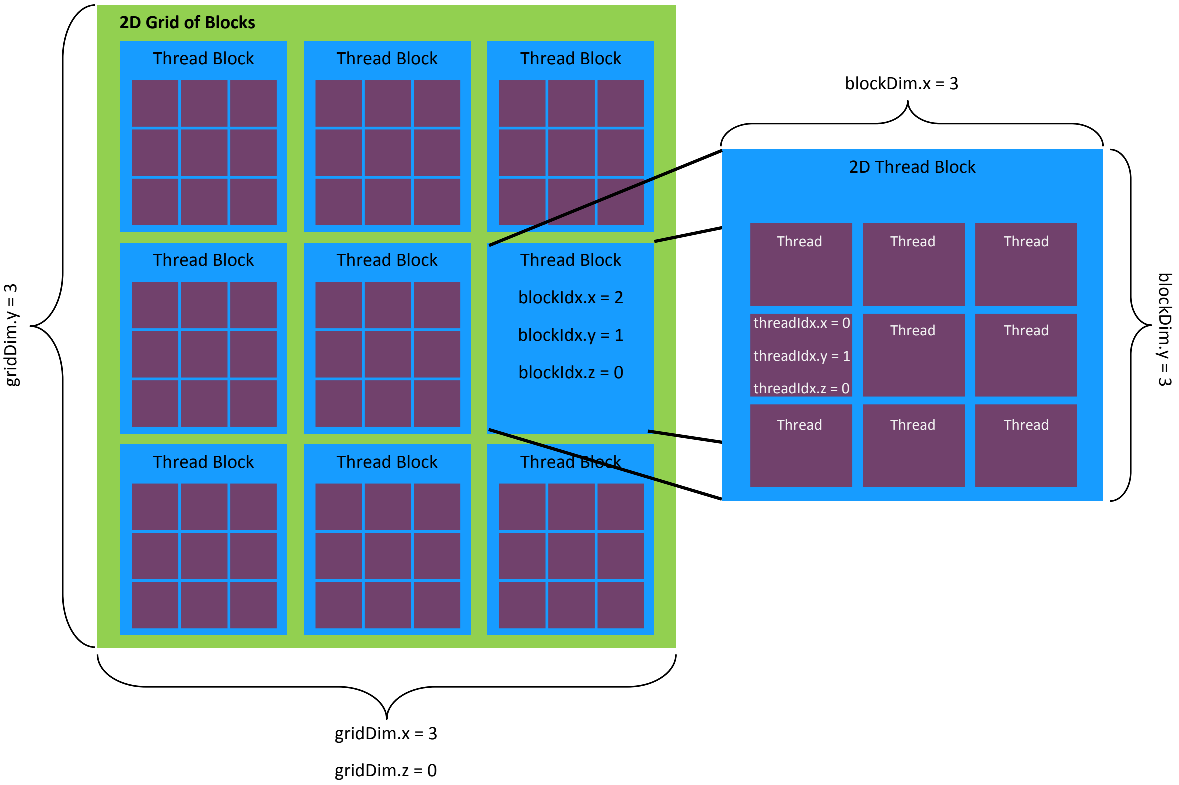


Figure 2.3: A grid used to launch a kernel, adapted from [85]. This is a 2D grid of blocks, where each block is a 2D block of threads. Each thread has a unique identifier labelled by the thread index within a block (specified by threadIdx) and the block index within the grid (specified by blockIdx).

3x3 grid where each block is a 3x3 thread block. In practice a grid of 32 blocks with 256 threads each would be launched for example so there would be 8192 threads in flight at once.

Each CUDA thread has a unique identifier used to access data, based on it's unique thread number within a thread block, as specified by the thread index (threadIdx), and it's unique thread block number within the grid as specified by the block index (blockIdx). For example, considering a 1D grid of blocks containing 1D blocks of threads each unique thread index is given by:

```
int tid = threadIdx.x + blockDim.x*blockIdx.x;
```

where `blockDim.x` specifies the number of threads in each block. The unique thread index label can then be used within kernels to access the data. For example, a simple kernel adding two long vectors is given:

```
__global__ void vectorAdd(int *a, int *b, int *c, int size){

  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  int i;

  for(i=tid; i<size; i+=blockDim.x*gridDim.x){
    c[i] = a[i] + b[i];
  }

}
```

When launching a kernel from host CPU code, the user specifies the number of threads per block and the number of blocks per grid to launch with. For example:

```
vectorAdd<<<32, 256>>>(a, b, c, size);
```

launches the `vectorAdd` kernel with 32 blocks each having 256 threads so there are 8192 threads in flight in total. In this way, CUDA gives users an intuitive way to exploit the underlying parallelism of the GPU hardware.

2.2 FPGAs

FPGA programming requires mapping the problem to the FPGA architecture and resources, such that most of the resources available on chip are utilized. FPGAs can provide effective acceleration when presented with a constant stream of data to be processed which keeps each logic element working every clock cycle.

FPGAs are essentially lots of reconfigurable logic blocks, meaning they can be designed to suit the application at hand which complements the structure of the application and the data in the problem. Unlike application-specific integrated circuits (ASICs) which are designed for a specific application and then fixed in this configuration permanently, FPGAs can be redesigned and configured for another problem with a different architecture. They also contain a variety of special purpose blocks such as internal memory, I/O blocks, and specialised arithmetic circuits. A unique feature FPGAs have is that they can be used as reconfigurable processors and can exploit fine grained parallelism tailored to a specific application [88].

In fusion applications, FPGAs have mostly been used in signal processing and data acquisition and have not really been used for high performance computing tasks yet. The dataflow engine was chosen over other FPGA based systems (such as Xilinx Vivado High-Level Synthesis) as the architecture to investigate in this thesis because it is a relative newcomer to the field and is marketed specifically for high performance computing on FPGAs.

2.2.1 Dataflow engines

Dataflow engines (DFEs) have been developed by Maxeler Technologies as an alternative to parallel programming for accelerating applications in a wide range of fields such as molecular mechanics simulations and the calculation of induced dipoles [89], sorting algorithms such as the bitonic mergesort algorithm [90], stencil computation known to be computationally demanding and prevalent in many scientific areas [91] and a specific finite difference library in the form of MaxGenFD [92]. Recently better energy efficiency of machines is also highly desirable as power and cooling costs are ever increasing and green computing is becoming a major concern for users and applications. The top 23 green supercomputers are based on heterogeneous architectures, using accelerator technology [93] so it is clear there is a shift towards energy efficient computing using emerging technology and the Maxeler dataflow engine can provide such a solution.

The programming paradigm for DFEs differs significantly from traditional CPU programming where a program's source code is transformed into a list of instructions for a particular processor which is then loaded into the memory attached to that processor [94]. Data and instructions are read from CPU memory into the processor core where operations are performed and results written back to memory as illustrated in Fig. 2.4. This model is inherently sequential and performance depends on the latency of memory accesses and CPU clock time.

In the dataflow scenario, the program source code is transformed into a dataflow engine configuration file which describes the operations, layout and connections of a dataflow engine. Data streams from memory into the FPGA chip where data is

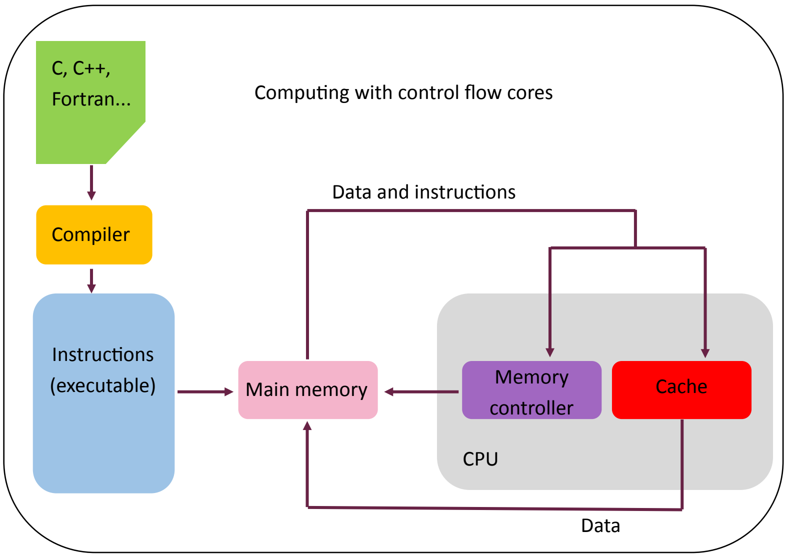


Figure 2.4: A schematic showing the instruction fetch and computation cycle over time in a CPU [94].

forwarded directly from one arithmetic unit to another, without being written to off-chip memory until the chain of processing is complete (Fig. 2.5). With a dataflow engine, the emphasis is to program in space meaning to effectively use all of the FPGA chip as all of the code (or operations) is executed simultaneously [94]. This is in contrast to the more usual models of programming in time where operations happen in order moving from one line of code to the next.

Fig. 2.6 shows the overall Maxeler system connecting the dataflow engine to the CPU via PCIe. The dataflow engine is programmed using a kernel (or many kernels) and a manager.

2.2.2 DFE software - MaxJ, kernel and manager code

DFE applications have three components: *(i)* the host code usually written in C, *(ii)* the kernel code and *(iii)* the manager code both written in MaxJ, an extension to Java. Kernels implement computation on the DFE whilst the manager organizes data movement to, from and within the dataflow engine. Together, the kernel and manager files are compiled with MaxCompiler [95] which generates dataflow implementations in the form of dataflow engine configuration (.max) files. This generates

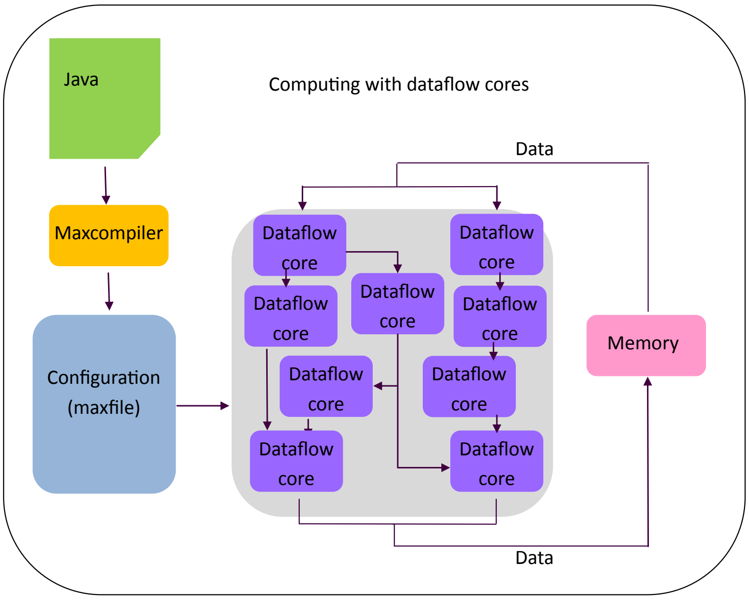


Figure 2.5: A dataflow program executing [94], indicating the streaming nature of computation and programming in space paradigm.

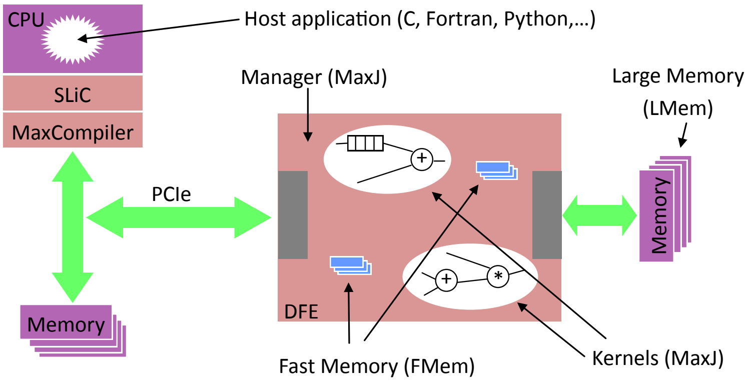


Figure 2.6: A diagram of the DFE system showing detail of the LMem (large off-chip memory), FMem (small on-chip memory), kernel and manager [95]. The DFE is connected to the CPU via PCIe.

a DFE function which can then be called from the CPU via the Simple Live CPU (SLiC) interface. The overall system is managed by MaxelerOS which sits within Linux. Kernel and manager code is written in MaxJ, removing the need for knowledge of VHDL or Verilog traditionally used for programming FPGAs. The Maxeler architecture, toolchain and programming model is described fully in [95], [96] and [97] and specifically for the finite difference application MaxGenFD in [98].

In section 2.1.2 a vector addition kernel for the GPU was given. Here a vector addition kernel for the DFE is shown:

```
package vectorAdd;

class vectorAddKernel extends Kernel {
  vectorAddKernel(KernelParameters parameters) {
    super(parameters);

    // Input
    DFEVar a = io.input("a" , dfeUInt(32));
    DFEVar b = io.input("b" , dfeUInt(32));

    // Addition
    DFEVar result = a + b;

    // Output
    io.output("c" , result , dfeUInt(32));
  }
}
```

This would then be called from the CPU as:

```
vectorAdd(size, a, b, c);
```

where size is specified in the manager code and is the length of the vectors. This parameter size in the manager also specifies the number of ticks (or clock cycles) for the DFE kernel to run for as a data element is read from a and b, the addition performed and the first data element of c output in the same clock tick.

2.3 Intel MIC co-processor

Another emerging technology is the Intel Xeon Phi Coprocessor which is a Many Integrated Core (MIC) architecture and uses legacy programming models such as OpenMP and MPI. This architecture is not studied in depth in the thesis as in a lot of cases, it's performance is not as promising as the GPU. For example, a one-dimensional electrostatic PIC code was ported to a MIC and GPU [99] but performance on the MIC was slower than on the GPU. Mention of the MIC is included here for completeness.

2.4 Comparison of the accelerating technologies

2.4.1 Performance comparison of example applications

The optimum technology to use is very dependent on the type of application or problem that needs to be accelerated. For example, Table 2.1 which summarises the time taken for a multi-threaded CPU, a GPU and an FPGA to perform a FFT benchmarch [100], appears to show that an FFT has the best performance on a GPU architecture. However, there is an overhead related to the data transfer between the CPU and GPU which limits the speed up. The data transfer overhead has not been taken into account. For the size of data used in this example the transfer time is 69ms so the GPU is actually slowest across the three platforms. Therefore it would appear this problem is best suited to an FPGA. The size of the problem needs to be large enough to compensate for the data transfer time to the GPU, or data transfer times need to be reduced. Although the FPGA now apparently has the fastest time for the FFT, there is no mention of the transfer time associated with the FPGA system in this study which can also be a bottleneck.

A comparison of the architecture and performance of a GPU and MIC is given in [101] and it is demonstrated that the MIC has similar performance, or even outperforms the GPU in some applications.

Fig. 2.7 shows a comparison for the implementation of gaxpy (a matrix-vector multiplication) implemented in C, Intel Math Kernel Library (with a single thread

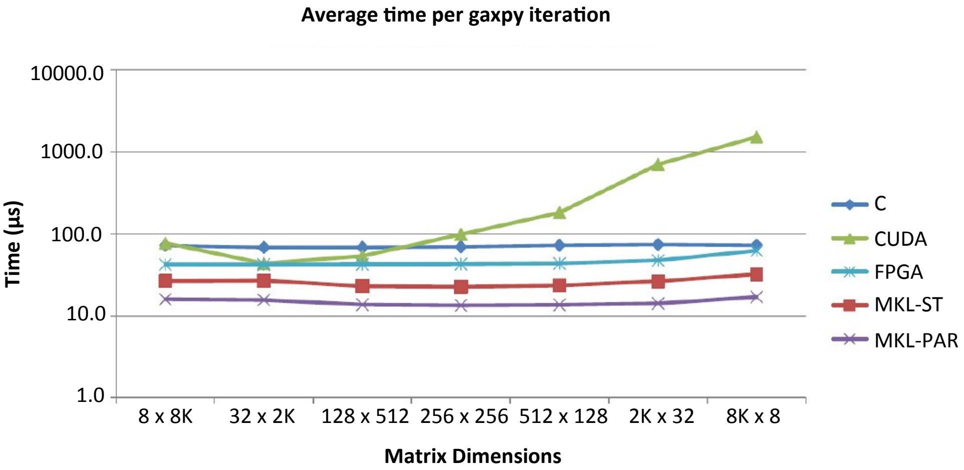
Table 2.1: FFT benchmark on different architectures from [100]

FFT benchmark 262155 points							
Device	CPU					GPU	FPGA
Threads	2	4	8	12	24	Max threads	Virtex-5
Time	76.17 ms	45.41 ms	31.63 ms	27.85 ms	31.36 ms	8.13 μs (execution)	2.59 ms

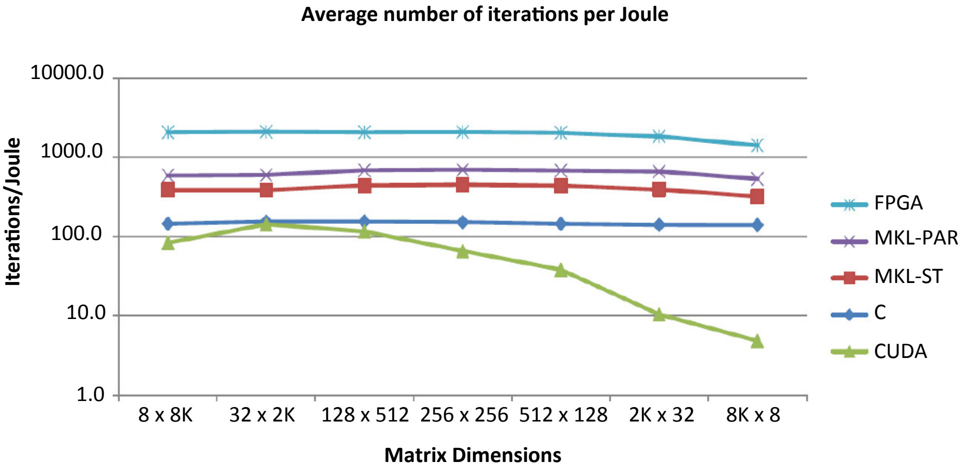
and in parallel), CUDA, and an FPGA implementation [102]. Performance results for small matrix sizes show that FPGA has similar performance (and at a higher energy efficiency) when compared to the CPU and GPU platform. A comparison of implementations of sum of absolute differences, 2D convolution and correntropy is shown in Fig. 2.8. This shows the FPGA often has the most acceleration over the C++ application, especially for large kernel size, with the GPU acceleration not as good. The FPGA is also the most energy efficient in these applications and the GPU has better energy efficiency than the CPU based applications.

2.4.2 Other factors to consider for a fair comparison

Whilst the most important factor to consider when choosing an architecture to use for acceleration of an application is the suitability of the architecture for the problem, and the expected performance or speed achieved, there are also other factors to consider. These include the cost of each device, the running cost or energy efficiency of each device and the programmability of each device. GPUs tend to be much cheaper than equivalent specification FPGA boards and are certainly much cheaper than the bespoke Maxeler dataflow engine architecture, but FPGAs are more energy efficient as they run at a slower clock speed. Energy efficiency is becoming an ever more important factor to consider with increasing running costs. However, FPGAs have traditionally been known to be difficult to program, with users facing a steep learning curve to achieve a successful acceleration of an application. Development time is significantly higher when porting to the new dataflow architecture when compared to porting an application to a GPU as GPU computing is a more mature

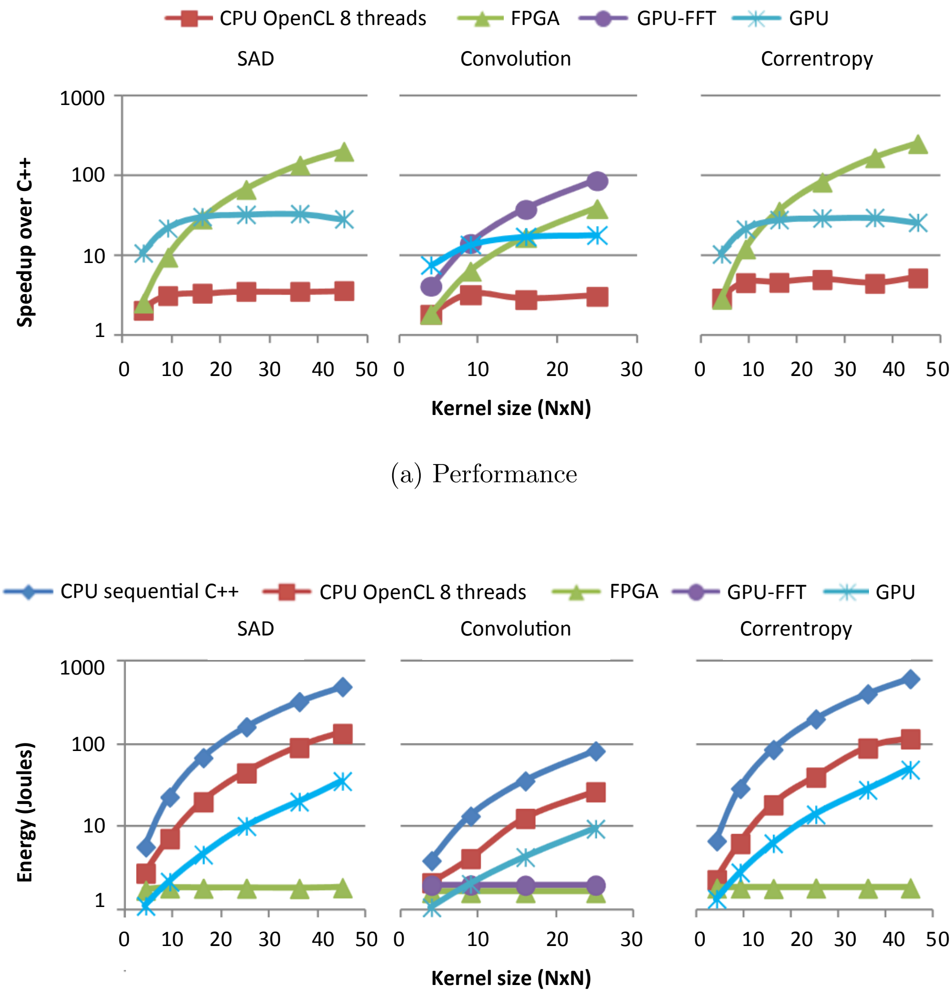


(a) Performance



(b) Energy efficiency

Figure 2.7: Comparison of BLAS Level 2 on CPU, GPU and FPGA in terms of (a) performance and (b) energy efficiency [102]. The GPU has the worst performance particularly for larger problem sizes, and the worst energy efficiency. This is due to the system and driver overhead for short and fat matrix dimensions and short vector effects for long and thin matrices. The FPGA has better performance and is by far the most energy efficient.



(a) Performance

(b) Energy efficiency

Figure 2.8: Comparisons of (a) performance and (b) energy efficiency tested on 720p images on CPU, GPU and FPGA [103]. The FPGA implementations have the highest acceleration whilst being the most energy efficient.

field and there is a lot of support for the technology. Potential users need to weigh up these options when deciding which technology to use. For example, if having a low running cost is the most important factor, then this comes at the expense of a longer development time. However, if an accelerated implementation is required immediately it might make more sense to choose a GPU. There is no "right" answer to the question "which is the best technology to use?": it is a complicated question with many factors to consider and weigh against each other.

To summarise, it is clear that these emerging architectures have the potential to provide large acceleration of applications, over traditional CPU-only applications. As experiments and simulations get larger, the use of these new technologies will become essential. There are however decisions to be made as to which technology to use with cost, availability and ease of use affecting the decision. As GPU technology is cheaper, more widely available and easily programmed, the thesis focuses mostly on the use of GPU technology for massive data-processing tasks such as those found with the Synthetic Aperture Microwave Imaging diagnostic (SAMI) and simulations of the burn-up and Bateman equations found in neutronics applications. This is covered in the following two chapters. In the final chapter, the use of FPGA technology in the form of the Maxeler dataflow engine is investigated as an energy efficient alternative in the context of a large dense matrix multiplication problem which is a common feature of many applications not only within the fusion community but more generally across all areas of scientific computing.

Chapter 3

GPU-based data processing for the SAMI diagnostic

In this chapter we present an application of GPU hardware acceleration to the processing of data from the synthetic microwave imaging (SAMI) diagnostic [104]. The necessity for an accelerated GPU code is highlighted and the GPU CUDA techniques presented, along with the acceleration results. Finally the multi-shot data analysis that has been enabled due to the accelerated processing of raw data by the GPU is discussed. Section 3.1 introduces the principles behind synthetic aperture microwave imaging and section 3.2 describes the SAMI diagnostic on MAST. Section 3.3 discusses the motivation for a GPU code to process the data before going on to discuss the GPU code in section 3.4. The acceleration results and the accuracy of the GPU code are presented in section 3.5 and a brief cost-performance analysis is carried out in section 3.6. Finally, in section 3.7 the multi-shot data analysis is carried out. Different models have been explored, based on an understanding of underlying physical processes which may be occurring, to try to explain the observed correlations between electron cyclotron and deuterium-alpha emission.

3.1 Synthetic aperture microwave imaging

Low frequency electromagnetic waves in a plasma cannot propagate in tokamaks like MAST, which have a high density, and so cannot be detected outside of the plasma. There are however thermally induced electrostatic waves in the plasma called electron Bernstein waves [105] which can convert to electromagnetic modes and escape the plasma [106]. SAMI can detect these waves at certain frequencies with an array of antennas and use the signals to reconstruct images of the mode conversion process. This is useful as the mode conversion process contains information about local plasma parameters such as the magnetic field structure [107]. Two types of electromagnetic wave propagation considered here are ordinary (O) mode and extraordinary (X) mode. The O-mode is linearly polarised with the electric field and the X-mode is elliptically polarised with a component of the electric field perpendicular to the magnetic field. There are two types of mode conversion that can occur: conversion of the Bernstein mode into the X-mode directly or conversion of the Bernstein mode into the ordinary O-mode via the X mode. However, the opposite processes are usually considered, where electromagnetic waves are launched into the plasma for heating purposes. The direct tunnelling from X-mode to Bernstein wave [108], and the conversion of the O-mode to X-mode [109] followed by the X-mode to Bernstein wave [110] have been studied widely [111]. By the reciprocity theorem [112], the inverse processes hold and SAMI can image the mode conversion process.

The O and X modes are the modes for propagation perpendicular to the magnetic field. The dispersion relations for the O and X modes are:

$$c^2k^2 = \omega^2 - \omega_p^2 \qquad (O - mode) \tag{3.1.1}$$

$$c^2k^2 = \omega^2 - \omega_p^2\left(\frac{\omega^2 - \omega_p^2}{\omega^2 - \omega_{uh}^2}\right) = \frac{\omega^4 - (\omega_{uh}^2 + \omega_p^2)\omega^2 + \omega_p^4}{\omega^2 - \omega_{uh}^2} \qquad (X - mode) \tag{3.1.2}$$

where $\omega_{uh}^2 = \omega_p^2 + \omega_c^2$ and ω_{uh} is the upper hybrid frequency which is a plasma oscillation where the electrons have elliptical orbits that are a combination of the plasma oscillation and the cyclotron oscillation. The plasma frequency, ω_p, and electron cyclotron frequency, ω_c, are defined by:

$$\omega_p^2 = \frac{n_e e^2}{m_e \epsilon_0} \qquad \omega_c = \frac{eB}{m} \tag{3.1.3}$$

A resonance occurs when $k \to \infty$ meaning the wave is absorbed and a cut-off occurs when $k \to 0$ meaning the wave is reflected. Therefore the O-mode has a cut-off at $\omega = \omega_p$, the X-mode has a resonance at $\omega = \omega_{uh}$ and two cut-offs at $\omega = \omega_{l,h}$ for the low and high density cut-offs where:

$$\omega_{l,h} = \left(\omega_p^2 + \frac{\omega_c^2}{2} \pm \omega_c \sqrt{\omega_p^2 + \left(\frac{\omega_c}{2}\right)^2} \right)^{\frac{1}{2}} \tag{3.1.4}$$

which follows from equation 3.1.2 with $k = 0$ implying:

$$\omega^4 - (\omega_{uh}^2 + \omega_p^2)\omega^2 + \omega_p^4 = 0 \tag{3.1.5}$$

Solving the bi-quadratic formula for ω^2:

$$\omega^2 = \frac{1}{2}\left(\omega_{uh}^2 + \omega_p^2\right) \pm \frac{1}{2}\sqrt{\left(\omega_{uh}^2 + \omega_p^2\right)^2 - 4\omega_p^4} \tag{3.1.6}$$

and substituting $\omega_{uh}^2 = \omega_p^2 + \omega_c^2$, the result in equation 3.1.4 follows.

The cut-offs and resonances can be seen in Fig. 3.1 which plots the dispersion relations for the O-mode and X-mode. The O-mode and X-mode cannot propagate in the plasma because of the cut-off surfaces. The electrostatic Bernstein waves inside the plasma convert to electromagnetic X-mode at the upper hybrid resonance when the upper hybrid resonance is close to the low density cut-off via tunneling. Alternatively, at the upper hybrid resonance the Bernstein wave converts to X mode and propagates to the high-density cut-off and then to the plasma density cut-off where it converts to an O mode which can then be detected outside of the plasma. The mode-converted Bernstein waves escape the plasma where they can be detected and images formed of the mode conversion window.

The images are formed by applying the van Cittert-Zernike theorem [113] [114] for an incoherent source in the far-field:

$$\Gamma_{i,j} = \int G(\eta,\xi) I(\eta,\xi) e^{ik(u\eta + v\xi)} d\eta d\xi \tag{3.1.7}$$

where $\Gamma_{i,j}$ is the cross-correlation function, $G(\eta,\xi)I(\eta,\xi)$ is the intensity distribution, $\eta = \cos\theta \sin\phi$ and $\xi = \sin\theta$ are the direction cosines of an antenna on a distant source, $u = \frac{b_x}{\lambda}$ and $v = \frac{b_z}{\lambda}$ are the components of a unit vector between

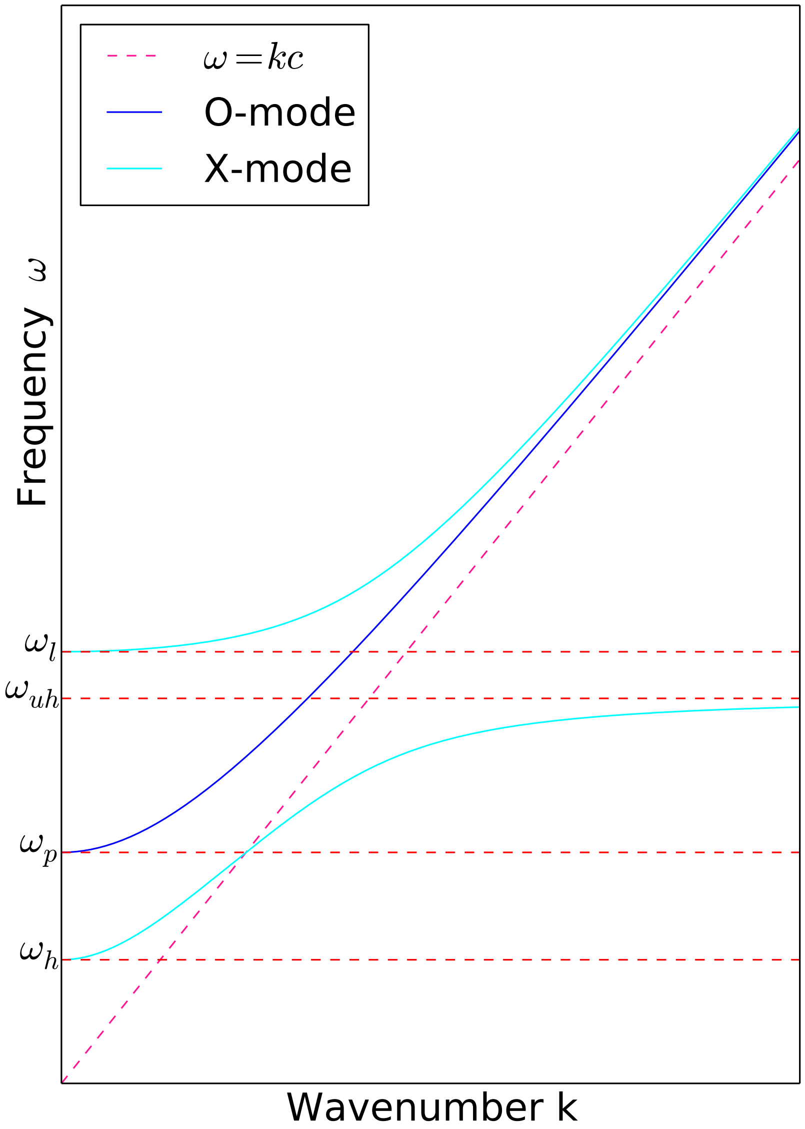


Figure 3.1: Dispersion relations for O and X modes indicating the cut-off frequencies ω_l, ω_p and ω_h, and the upper hybrid resonance ω_{uh}.

antenna pairs with λ being the vacuum wavelength. Equation 3.1.7 can be seen as the Fourier transform of the brightness distribution of the source or the mutual coherence function [104]. This is equivalent to the cross-correlation function:

$$\Gamma_{i,j} = \int s_i s_j^* dt \tag{3.1.8}$$

where s_i is the total power (voltage) received in SAMI antenna i [115] defined by:

$$s_i = \int g(\eta,\xi)E(\eta,\xi)e^{ik(a_{x,i}\eta + a_{z,i}\xi)}d\eta d\xi \tag{3.1.9}$$

where $g(\eta,\xi)$ is the voltage received by the antenna and $a_{x,i}$ and $a_{z,i}$ describe the location of the antenna. Measuring the voltage received by each antenna in the array, the cross-correlations between each antenna pair can be calculated and applying the van Cittert-Zernike theorem, performing an inverse Fourier transform to reconstruct the image. With the eight antenna SAMI configuration, there are 28 independent antenna pairs ($\texttt{nAnt}(\texttt{nAnt}-1)/2$) for which the cross-correlations need to be calculated [104]. This gives enough samples in Fourier space so that an inverse Fourier transform can be used to reconstruct the brightness distribution image from the cross-correlations.

3.2 Description of the SAMI diagnostic

The SAMI diagnostic has the highest data rate of any diagnostic on MAST. For a typical shot on MAST, the amount of data produced by diagnostics excluding SAMI amounts to approximately 120-140 MB and the images produced by diagnostics excluding SAMI account for approximately 350 MB. SAMI scans over 16 frequency channels in the range 10 GHz-35.5 GHz and the signals are digitised by 14-bit ADCs sampling at 250 MSPS, giving a data rate of 8GB/s. So for a single 500ms shot on MAST 4GB raw data is acquired [116]. MAST aims for 30 shots a day meaning SAMI acquires 120 GB/day making SAMI a good test case for future high data rate diagnostics and the associated data handling expected for ITER [66], as discussed in Chapter 1.

The SAMI system installed on MAST is a phased array of eight linearly polarized Vivaldi antennas with a configuration that has been optimized to achieve maximum synthetic aperture efficiency satisfying both space and bandwidth requirements [117]. For typical MAST parameters, thermally born electrostatic electron Bernstein waves (EBWs) are emitted anisotropically and coplanar with the density gradient and the magnetic field at the mode conversion surface. Since the density gradient is known from other diagnostics, SAMI can be used to deduce the magnetic field line pitch. SAMI operates in two modes simultaneously: the passive imaging mode detecting spontaneous EBW emission, and the active probing mode measuring the back-scattered signal from a probe source. The data is acquired on two ADC boards (each digitizing eight channels) by two FPGA boards running embedded Linux for control. The data is streamed as a series of UDP packets over ethernet from the FPGAs via fibre optic cable to a file on the data storage computer. A schematic of the SAMI data acquisition system is presented in Fig. 3.2 showing the plasma on the left and both the active probing signal (indicated by the solid arrow) and spontaneous emission from the plasma (indicated by the dashed arrows). SAMI has produced the first ever 2D thermal electron Bernstein emission (EBE) maps of a plasma, identifying the location of B-X-O mode conversion windows in over-dense plasmas. SAMI is also the first diagnostic to measure magnetic pitch angle through simultaneous 2D Doppler backscattering [118]. Fig. 3.3 is the image reconstruction for shot 27022 at 260ms showing the brightness distribution or microwave intensity as a function of angular position. The location of the mode conversion windows is clearly identified. The images are reconstructed from the cross-correlations between each pair of antenna signals which are calculated by the data processing code post shot.

3.3 Motivation for GPU data processing code

SAMI acquires 4GB raw data per shot and an existing Interactive Data Language (IDL) data processing code which calculates the cross-correlations between antenna pairs takes approximately 20-30 minutes per shot to run and as such is unsuitable

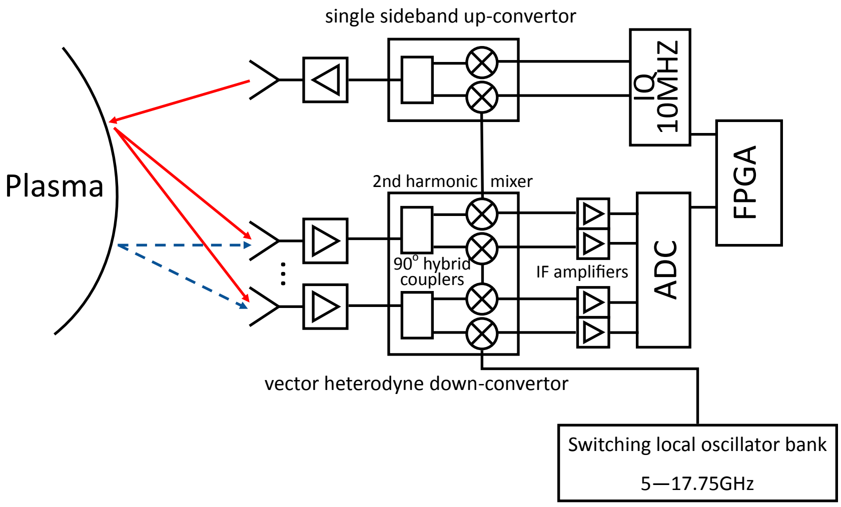

single sideband up-convertor
IQ
10MHZ
FPGA
Plasma
2nd harmonic mixer
90° hybrid couplers
IF amplifiers
ADC
vector heterodyne down-convertor
Switching local oscillator bank
5—17.75GHz

Figure 3.2: A schematic of the SAMI data acquisition system showing both active probing (solid arrows) and passive imaging (dashed arrows) modes.

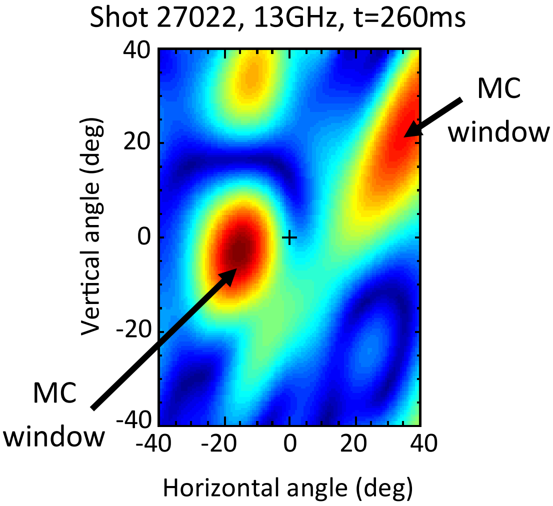


Figure 3.3: A typical image reconstruction showing the brightness distribution for shot 27022 at 13 GHz and 260ms into the shot. The location and size of B-X-O mode conversion (MC) windows are clearly identified by peaks in the brightness distribution.

for inter-shot processing. Faster data processing would allow the SAMI images to be examined between shots and used to update parameters of the next shot accordingly. To reduce the image reconstruction time, a GPU-based Compute Unified Device Architecture (CUDA) code has been developed demonstrating a significant acceleration of the data processing, making it possible for inter-shot processing in future campaigns on the National Spherical Torus Experiment (NSTX-U) [119] and MAST-U [120] thus greatly improving the capabilities of the SAMI diagnostic on these machines. Further, the GPU code will enable data-mining of many MAST shots from previous campaigns which has until now been impossible due to the long runtime of the existing IDL code.

For the M8 (2011 and early 2012) and M9 (2013) campaigns on MAST, raw data for approximately 3000 shots was obtained, requiring a 12TB RAID data storage system. On this system, with an AMD Phenom(TM) II X2 560 processor, it takes approximately 30 minutes to calculate the cross-correlations for one shot with the existing IDL code. MAST operates on a cycle of 8 hours on, 16 hours off and aims to carry out 30 shots per day. Therefore with the IDL code, SAMI data could be processed in 15 hours. In reality, the raw data for SAMI was not processed overnight as it was acquired but was stored for processing at some later date. Despite the length of data acquisition on MAST being relatively short (a MAST pulse is 500 milliseconds) and the time between consecutive pulses on MAST being relatively long (between 15 and 20 minutes in most cases), the IDL data processing code has a significantly longer runtime. For devices like MAST-U and future devices like ITER with much longer pulse lengths, this processing time will increase further so it is essential to dramatically speed up the processing time. The accelerated GPU code presented below has enabled overnight data processing. More impressively, in the future, data can be processed between shots on MAST-U moving into a new regime of inter-shot data processing.

Data mining SAMI multi-shot cross-correlation data could also help derive scaling laws; for example early analysis indicates a dependence between electron Bernstein wave power and deuterium-alpha D_α emission, but it is essential to investigate multiple shots to find correlations between plasma parameters. Many-shot analysis has previously proved to be vital in the derivation of scaling laws such as ITER98Y2 [121] which is now the commonly used model for ITER design. This is based entirely on empirical scaling from regression analysis of data in the ITPA [122] database which includes a large number of shots from a number of different tokamaks.

3.4 GPU code

3.4.1 Data processing requirements

The raw data SAMI collects needs extensive processing to: (*i*) correct for phase drift between in-phase (I) and quadrature (Q) components of the data signal, x (which is expressed as $x(t) = I(t) + iQ(t)$ so that I is the real part of the signal and Q is the imaginary part), (*ii*) perform sideband separation and (*iii*) correct for phase differences between antennas due to RF electrical lengths. Sideband separation results in two sidebands, the upper sideband and the lower sideband which have opposite phases making the measurement of the phase difficult. However, sideband suppression can completely suppress either the upper or the lower sideband by shifting either sideband signal by 90° in phase so that only the other sideband remains. Better sideband suppression is obtained if the phase dispersion between I and Q signals introduced by the IF signal cables and filters is corrected first. Once the sidebands are successfully separated, the phase difference between antenna channels can be corrected. This phase is affected by differences in path lengths giving rise to unknown phases between the RF signals. Once the raw data has been corrected for these effects, the cross-correlations between each antenna pair are calculated from which the images can be formed. This process is shown schematically in Fig. 3.4 with the most significant system bottlenecks enclosed in circles. The Fourier filter operations and the cross-correlation steps dominate the data processing time.

The numerical operations carried out in the data processing chain are presented in Table 3.1 for a typical MAST shot and the data is processed as a series of many independent vectors for acceleration processes. At the start of the processing chain we have 16 real signals and for a typical MAST shot 16 frequency channels and

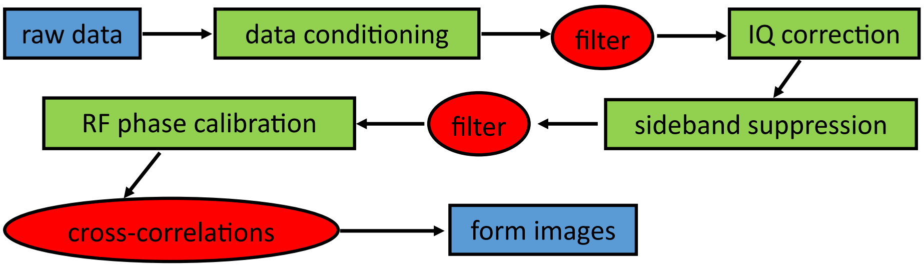


Figure 3.4: The data processing chain for SAMI data. Computationally expensive calculations are enclosed in circles and include the Fourier filter and cross-correlation steps.

3119 time sweeps each containing 2500 time points. After extracting the middle of each time sweep to remove noise introduced by switching frequency channel, each time sweep contains 2000 points. With sideband separation 16 real signals get converted to 8 complex signals and there are now two data arrays for the upper and lower sidebands. There are $\texttt{nAnt} \times (\texttt{nAnt} - 1)/2 = 28$ unique cross-correlations to calculate where $\texttt{nAnt}$ = number of antennas, and therefore there are 28 associated baselines. The image is then formed by aperture synthesis as the sum of the products of antenna cross correlations and associated basis functions.

3.4.2 Suitability of SAMI for GPU acceleration

The suitability of the SAMI data processing code for parallelization by a GPU and CUDA is demonstrated in Fig. 3.5. The data is organised as a series of vectors of length `nInt` for each of the 8 antennas and each of the 16 frequency channels, `nf`, which are switched between `nSweeps` times. Essentially, the time series is split into `nSweeps` blocks of `nInt` points and each block of `nInt` points is operated on identically. As seen in Table 3.1, for a typical MAST shot where `nInt` = 2000, `nf` = 16 and `nSweeps` = 3119, the numerical computations consist of almost 800,000 vector operations of length 2000 elements being operated on identically, and the cross-correlation calculation consists of almost 2,800,000 vector operations. This is a SIMD scenario which is ideal for parallelization on a GPU using CUDA. Threads are grouped into thread blocks on the software level, typically each thread block represents a new vector and threads in the thread block represent individual data

Table 3.1: Numerical operations and typical size

PHYSICS OPERATION	MATHEMATICAL OPERATION	TYPICAL VECTOR LENGTH	TYPICAL NUMBER OF VECTORS
data conditioning	box-car average smooth	2500	798464
	extract middle	2500 ->2000	798464
	shift (not applied to all channels)	2000	199616
filter	FFT	2000	798464
	bandpass filter	2000	798464
	IQ filter	2000	798464
	inverse FFT	2000	798464
IQ correction	vector-scalar multiplication	2000	798464
sideband suppression	vector addition	2000	399232
	copy (for upper and lower sideband)	2000	399232
filter	FFT	2000	798464
	bandpass filter	2000	798464
	inverse FFT	2000	798464
RF phase calibration	complex vector scalar multiplication	2000	798464
cross-correlations	vector mean	2000	798464
	subtract vector mean	2000	798464
	vector dot product	2000	2794624

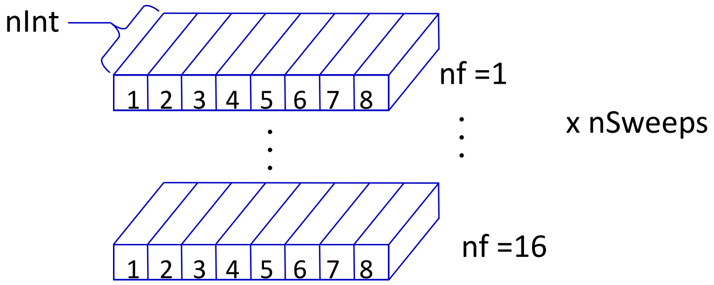


Figure 3.5: A schematic of the SAMI data structure highlighting the SIMD nature. The data is processed as a series of vector operations on vectors of size `nInt` where `nInt` is the number of data points collected before SAMI switches to the next frequency channel.

points in the vector. CUDA kernels are then launched with multiple thread blocks and multiple threads per block to process the data simultaneously. A typical kernel launch would include 32 thread blocks each with 256 threads.

3.4.3 Description of the GPU code

The raw data is stored in binary files where the frequency channels are multiplexed in time. The raw data is read by demultiplexing the different frequency components by looping over each frequency channel, `nf`, and sweep, `nSweeps`, reading `nInt*nAnt` points from the two data files (one for channels 0-7, one for channels 8-15) and placing it in the correct location in the data array for efficient processing. Since the switching period and frequency order may vary from shot to shot, a `bootconFig.rfctrl.ini` file is consulted which informs where to start reading the binary file from on each iteration. The data is then copied to the GPU and the necessary signal processing tasks and data conditioning are performed in the following sequence. The noise of the local oscillator switch is first removed using a smoothing box-car average on each set of `nInt` points; the first and last 250 data points in each set of `nInt` points is discarded to avoid any residual switching noise when switching to the next frequency channel. The data corresponding to channel 0, channel 1, channel 8 and channel 9 is then shifted to correct for ADC timing errors. Finally, the data is filtered with a bandpass and notch filter to remove any unwanted

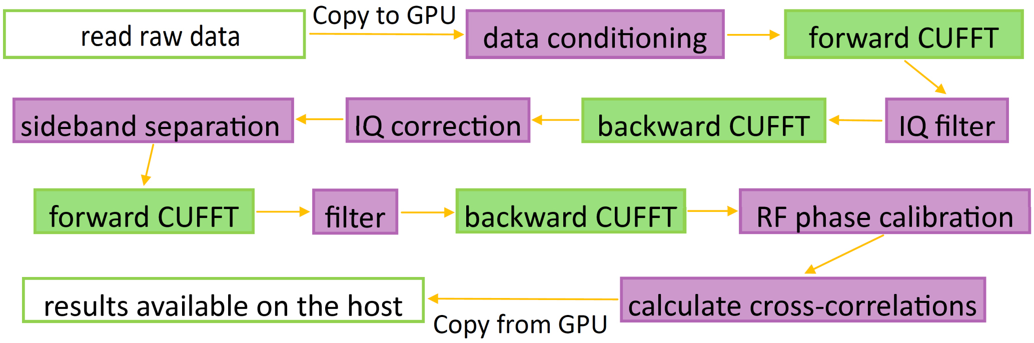


Figure 3.6: A schematic of the SAMI CUDA data processing code showing an initial data copy to the GPU, all data processing performed on the GPU and finally a resulting data copy back to the CPU.

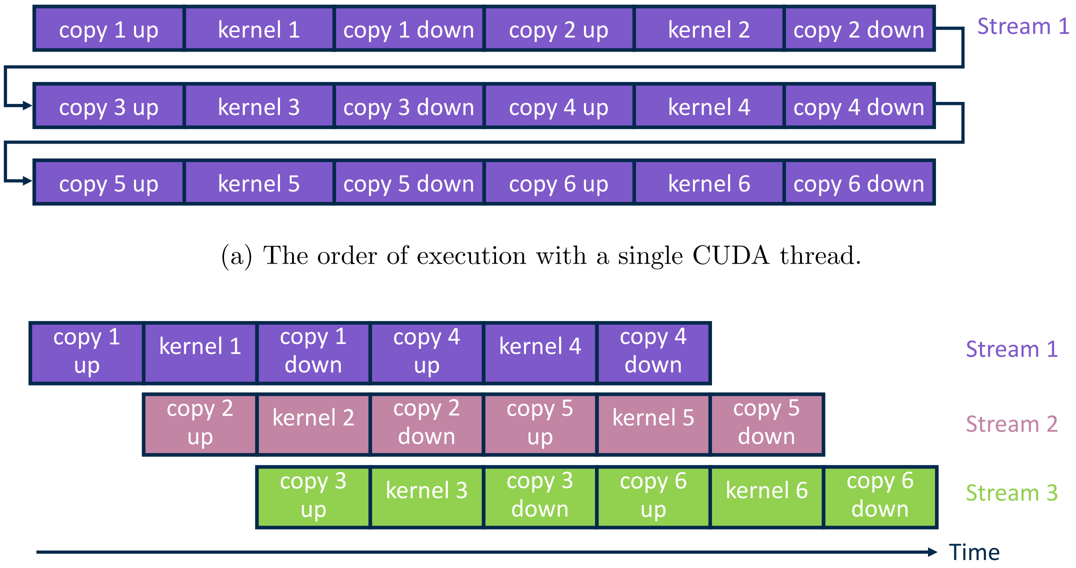
signals from each block of `nInt` data points and the IF dispersion between I and Q components caused by cable lengths is corrected. Performing sideband separation converts 16 real signals into 8 complex signals and another filter is performed for upper and lower sidebands, with calibration data correcting for phase offsets and balancing amplitudes between the I and Q components. Calibration of the RF phase is also performed to correct for any phase drifts in the RF channels after sideband suppression. Once these corrections have been made, the cross-correlations between the signals for each antenna pair, frequency sweep and upper and lower sideband are calculated. This overall process is illustrated in Fig. 3.6. In total, 14 CUDA kernels were constructed (see Appendix A) to process the data and the CUFFT library was used to perform the Fourier transforms.

As indicated in Fig. 3.4, the most computationally expensive parts of the program are performing the Fourier filtering, and the cross-correlation calculation itself. However, a lot of the initial data conditioning, as operations on a set of vectors, is ideal to target with a GPU. The data is therefore transferred to the GPU as soon as possible, all the processing performed on the GPU and the result copied back to the CPU once all the processing has completed, with no intermediate data traffic. Data movement is often the bottleneck to hybrid CPU-GPU computation and is hence avoided here.

3.4.4 CUDA streams and concurrency

There is however a problem with the above model for the SAMI computation, and for any large data problem such as those expected in the next generation of fusion diagnostics. GPUs have limited memory available and very few models have enough to process an entire SAMI shot at once. Each SAMI shot is 4GB of 14 bit (or two byte) integers. Immediately in software, the amount of data is doubled as each two byte integer is converted to a four byte float to perform scientific computation, so each SAMI shot is actually an 8GB data processing problem. It is therefore necessary for big data problems like SAMI to carve the data up into chunks and process each separately, exploiting CUDA streams and concurrency as a means to overlap a memory copy to the GPU with kernel execution on the GPU. The GPU can be effectively kept busy by copying the next chunk of data to the GPU whilst computing on the previous chunk. High-end GPUs such as the Tesla K40C can do even better as these GPUs have two copy engines to facilitate bi-directional memory copies. Therefore chunk$_{i-1}$ can be copied from the GPU, whilst computation is being performed on chunk$_i$ and the copying of chunk$_{i+1}$ to the GPU, is occurring simultaneously. This advantage is illustrated in Fig. 3.7 for an idealized scenario where the copy time and kernel execution time is assumed to be equal. Instead of having a single CUDA stream where successive data chunks are processed serially (although the data in each chunk is processed in parallel), if there are multiple CUDA streams, each with their own instruction queue, memory copies and kernel execution can be overlapped between streams and the time taken to process the data can be significantly reduced.

There is a balance between the number of streams used and the size of each data chunk, as each stream needs its own memory to hold different data chunks simultaneously but ideally as much data as possible should be processed at once so there are fewer data chunks. For example, with SAMI, if the whole 8GB (which doesn't fit in the GPU memory) was carved up into four data chunks each having size 2GB and there were four CUDA streams processing the data, then the memory requirements would still be 8GB and this scenario would not work as the GPU does not have enough memory for all of the streams. In practice, SAMI used three CUDA



(a) The order of execution with a single CUDA thread.

(b) The benefit of exploiting three CUDA streams and concurrency is illustrated.

Figure 3.7: A schematic showing (a) the order of execution using a single CUDA thread where the total time to process the data is 18 units of time, and (b) using three CUDA streams to process the data reduces the units of time required to process the data to eight.

streams and the size of each data chunk depends on the parameters `nInt`, `nf` and `nSweeps` which vary from shot to shot due to the switching period and frequency order varying from shot to shot. Typically the data is carved up into many smaller chunks and the number of data chunks would be on the order of 100 chunks.

3.5 Results

Code development was carried out on a machine with an Intel (R) Xeon(R) CPU E5-2670 @ 2.60 GHz with a Tesla K40C GPU with 12GB GDDR5 memory and PCIe 3.0 x16 lanes, which is independent to the SAMI data storage machine. The data for a few shots was made available on this machine to verify the code and obtain some preliminary acceleration results, shown in Table 3.2. The original IDL code averaged a time of 17 minutes and 18 seconds to calculate the cross-correlation data for a full shot. A serial C version developed as an intermediate step to the CUDA version was able to complete in 7 minutes and 44 seconds, giving an acceleration of 2.2x over

Table 3.2: Acceleration Results

			CUDA			
			Tesla K40C		**GeForce GTX770**	
	IDL	**C**	/dev/shm	hard drive	/mnt/ramdisk	hard disk
Total time (s)	1038.38	464.55	17.42	76.02	25.44	70.34
Acceleration		**2**	**59**	**13**	**40**	**14**

the IDL version of the code. The raw data for both the IDL and C implementations was loaded in `/dev/shm` shared memory in the form of a RAM disk. Accelerating further with CUDA on the Tesla K40C gave an execution time of just 17 seconds, an acceleration of 26x over the serial C and 59x over the original IDL. The total run time of 17 seconds for the Tesla K40C is for raw data loaded in `/dev/shm`. If the data is retrieved from a conventional hard drive, the run time of the code is significantly increased to 76 seconds. After demonstrating this acceleration on the Tesla K40C, a dedicated GPU card was obtained for SAMI which was a GeForce GTX770 with 4GB GDDR5. Running the code and retrieving the raw data from hard disk gives a runtime of 70 seconds for one shot but if we mount the data in a RAM disk, again the run time is significantly reduced to 25 seconds which is comparable to the Tesla K40C system. It is evident that as the GPU processes the data so quickly, retrieving the data from hard disk creates a serious bottleneck.

The CUDA times given in Table 3.2 are the total time taken for the code to run which can be further split up into CPU processor time and GPU processor time as shown in Table 3.3 for the GeForce GTX770. The time taken to read the raw data dominates the run time. The GPU time, including memory copies to and from the GPU, is consistent between the two cases but if the data is mounted in a RAM disk, the time taken to read all of the raw data from file is significantly reduced by 3.7x. The unaccounted for time contributing to the total run time is due to the setup of the correction data and filter functions used to calibrate and condition the data which is shot dependent. The GeForce GTX770 on the SAMI system was then used to cycle through the SAMI data for 1837 shots in 30 hours, averaging a total run time of 58 seconds per shot.

Table 3.3: Read times on SAMI system

	/mnt/ramdisk	hard disk
CPU read (s)	16.38	60.81
GPU incl. memcopies (s)	7.32	7.40
Total runtime (s)	25.44	70.34

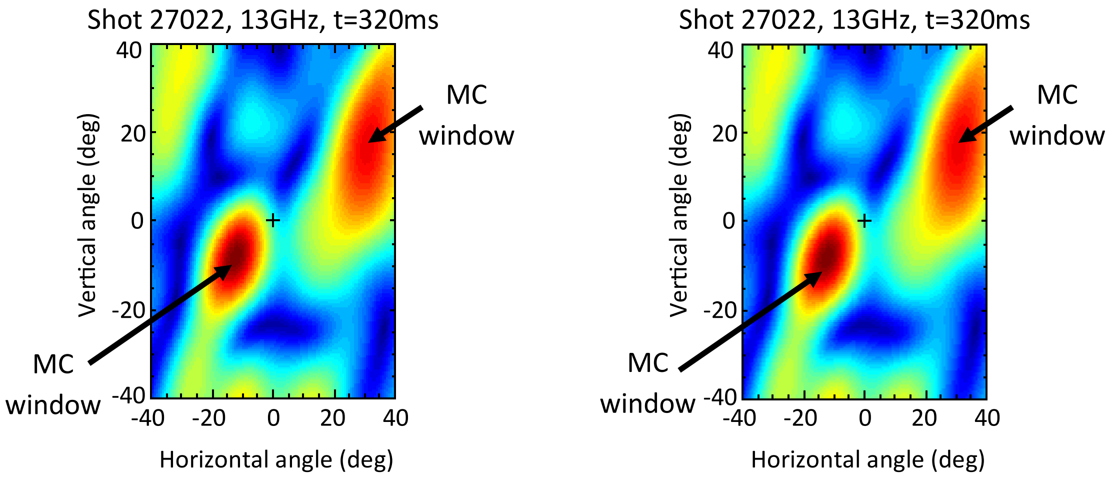
Table 3.4: Runtimes for different numbers of streams on Tesla K40C

	Number of streams			
	1	**2**	**3**	**4**
CPU read (s)	12.77	12.86	12.94	12.94
GPU incl. memcopies (s)	11.96	6.32	6.82	6.85
Total runtime (s)	24.89	19.39	20.01	20.06

Looking at Table 3.4 and Table 3.5 the benefit of using CUDA streams and overlapping kernel execution and memory copies is evident. Table 3.4 shows the execution times for a single stream implementation and multi-stream implementations on the Tesla K40C. For a single stream, the total GPU time including memory copies is 12 seconds whereas if multiple CUDA streams are used, the GPU time is nearly halved to between six and seven seconds. In this case, increasing the number of streams further provides no reduction to the GPU time as the GPU streaming multiprocessors, where computation is performed, are already fully utilized with two streams. So whilst there is space in memory for more streams, the computational resources are fully loaded with the kernels running for two streams. Table 3.5 shows the division of GPU time for the single stream case. The GPU time is dominated by the kernel execution and copying the result from the GPU is fast since we are reducing the amount of data by two orders of magnitude when calculating the cross-correlations. By referring to the multi-stream times in Table 3.4 it is clear that successful overlap of the memory copies has been achieved as the latency of the memory copies is hidden by the kernel execution time. The total GPU time including memory copies for multi-stream implementations is approximately equal to the kernel execution time for a single CUDA stream.

Table 3.5: Split of GPU time on the Tesla K40C for one stream implementation

	Time (s)
Copying to device	4.85
Kernel execution	6.94
Copying from the device	0.17



(a) Image reconstructed from the cross-correlations calculated by the IDL code.

(b) Image reconstructed from the cross-correlations calculated by the CUDA code.

Figure 3.8: The images reconstructed for shot 27022 at 13GHz and 320ms for (a) the IDL code and (b) the CUDA code . The images are in good agreement with relative error less than 10^{-4}.

Having achieved a nearly 60x acceleration over the IDL code, it is important to ensure the CUDA calculation is accurate. Using shot 27022 as an example, the cross-correlations were calculated using both the IDL code and the CUDA code. Fig. 3.8 shows the images of microwave emission on MAST at 13GHz and 320ms reconstructed from the cross-correlation data. The images produced for the IDL calculation and the CUDA calculation match well. In fact, the absolute error between the IDL cross-correlations and the CUDA cross-correlations is less than 10^{-8} and the relative error is less than 10^{-4}. The relative error is greater than the absolute error as some of the cross-correlation values are less than one.

3.6 Cost-performance analysis

It is important to emphasise that in Table 3.2 the C version is a serial implementation, utilizing a single core on a multi-core CPU. A fairer comparison would be to compare a parallel implementation utilizing all eight cores of the E5-2670 CPU using OpenMP. Assuming perfect scaling, using eight cores would give a time of 58 seconds and using 16 threads, two per core with Intel's hyper-threading would give a time of 29 seconds which is competitive with both GPU implementations. However, it is unlikely this kind of perfect scaling would occur and the multi-core times would be worse than this in reality so the GPU implementation maintains the best performance. Looking at the cost of each device, at launch in October 2013 the Tesla K40C with 12GB GDDR5 was \$7699 and has a double precision peak performance of 1.4 TFlops where as the E5-2670 had a launch price in March 2012 of \$1552 and peak performance of 166.4 GFlops based on a clock speed of 2.6 GHz, eight cores and eight instructions per cycle. In addition, the cost of 12GB DDR3 RAM must be included for the E5-2670 processor to make this a fair comparison; four 3GB cards can be purchased for approximately \$80. Therefore, the Tesla K40C has a slightly better performance per dollar of 0.181 GFlops/\$ compared to the E5-2670 system which has a performance per dollar of 0.102 GFlops/\$. However, since the launch of each device the price of the Tesla K40C has fallen considerably and if it was purchased today, it would cost approximately \$4600 where the cost of the E5-2670 has remained roughly constant and only fallen by \$100 or so. Therefore, at current prices, the Tesla K40C GPU has a much better performance per dollar of 0.298 GFlops/\$. For SAMI, since the amount of resulting data being copied back from the device is much reduced, the benefit of moving to the high-end Tesla K40C with bi-directional memory copies is small, as can be seen by comparing the GPU run times of the Tesla K40C and GeForce GTX770 in Table 3.3 and Table 3.4. The performance gain by moving to the Tesla K40C is only one second. However, the GeForce GTX770 with 4GB GDDR5 costs approximately \$460, an order of magnitude less than the Tesla K40C, so the performance per dollar for the GeForce GTX770 is an order of magnitude greater in the SAMI data reduction and analysis case.

3.7 Multi-shot analysis

In this section we use the GPU code presented in section 3.4 to process SAMI data from many plasma shots taken at MAST in the M8 (2011 and early 2012) and M9 campaigns (2013). The processed data for many SAMI shots is then able to be analysed for the first time.

3.7.1 Observed relationship between $\mathbf{D}_\alpha$ emission and ECE power

The accelerated GPU code has enabled the raw SAMI data from previous MAST campaigns to be processed and analysed for the first time. A relationship has been observed between the ECE power and the D_α emission in multiple shots. To observe this relationship it must be guaranteed that: *(i)* the reflectometer is off, *(ii)* it is a double null shot and *(iii)* it is a H-mode shot. MAST has a microwave reflectometer diagnostic [123] used for density profile measurements. The reflectometer needed to be off to obtain useful SAMI shots as it can affect the ECE power measurements. To tell if it was on or off, the SAMI data for the toroidal field (TF) test shots at the start of each day were analysed and it was determined from the ECE signal if the reflectometer was on for that day or not. Unfortunately, there are only 19 TF test shots (i.e. 19 days) for which SAMI data was also taken and of these, only 11 were guaranteed to have the reflectometer off for that day so this significantly reduced the number of shots available for analysis. An additional requirement that the shot be double null is so that we are observing on the midplane which removes the requirement for ray tracing, and a requirement that the shot be a H-mode shot is because the observed relationship between ECE power and D_α emission is occurring just before the shot enters H-mode.

The ECE power is extracted from the self-cross-correlations for each antenna, i.e. from the diagonal elements for each frequency and time slice. The D_α emission is the MAST signal `AIM_DA/HM10/T*`. The D_α signal has then been interpolated so that it is on the same time-base as the SAMI data. To identify the time windows of interest the data for each shot was looped over with increasing window size to find the window that had the best negative correlation, as measured by the Pearson correlation coefficient, between ECE power and D$_\alpha$ emission. If the best negative Pearson correlation coefficient was above a threshold then the shot was considered not to exhibit this relationship and was not included in this analysis.

The plots in Fig. 3.9 and Fig. 3.10 show eight different shots (26869, 27150, 27171, 27566, 27556, 27668, 28149, 28829 and 29144) with D$_\alpha$ emission on the x-axis and the natural log of the ECE power on the y-axis. The data points are colour coded with respect to time in the shot. These plots show there are time periods in each shot where the ECE power increases, in some shots dramatically, whilst the D$_\alpha$ emission decreases.

There appears to be a log-linear relationship between the ECE power and the D$_\alpha$ emission which can be expressed as:

$$ln(P_{ECE}) = GD_\alpha + c \Longrightarrow P_{ECE} = A * e^{GD_\alpha} \tag{3.7.1}$$

where $e^c = A$ and parameter c is the intercept and G is the gradient of the line of best fit drawn through each data set for each shot. The aim is to relate these parameters, particularly G, for each shot to plasma parameters to discover a general relationship for multiple shots which explains this observed correlation between the ECE power and D$_\alpha$ emission.

3.7.2 Plasma parameters effecting ECE power

The ECE power detected depends on properties of the mode conversion window like the thickness of the layer or mode conversion scale length L_{MC} and the temperature T_e [124] [125]. If L_{MC} is thick, less ECE power will be received at the antenna as the emission can not escape the plasma as easily. So over the time period shown, L_{MC} is expected to decrease as P_{ECE} increases. If T_e is high, the electrons have more energy and so it is easier to penetrate the mode conversion (MC) layer and escape the plasma and so more ECE power will be detected at the antenna. Additionally a higher temperature plasma is more ionized which suppresses D$_\alpha$ emission. So over the time period shown in the plots T_e is expected to increase as P_{ECE} increases and D_α decreases.

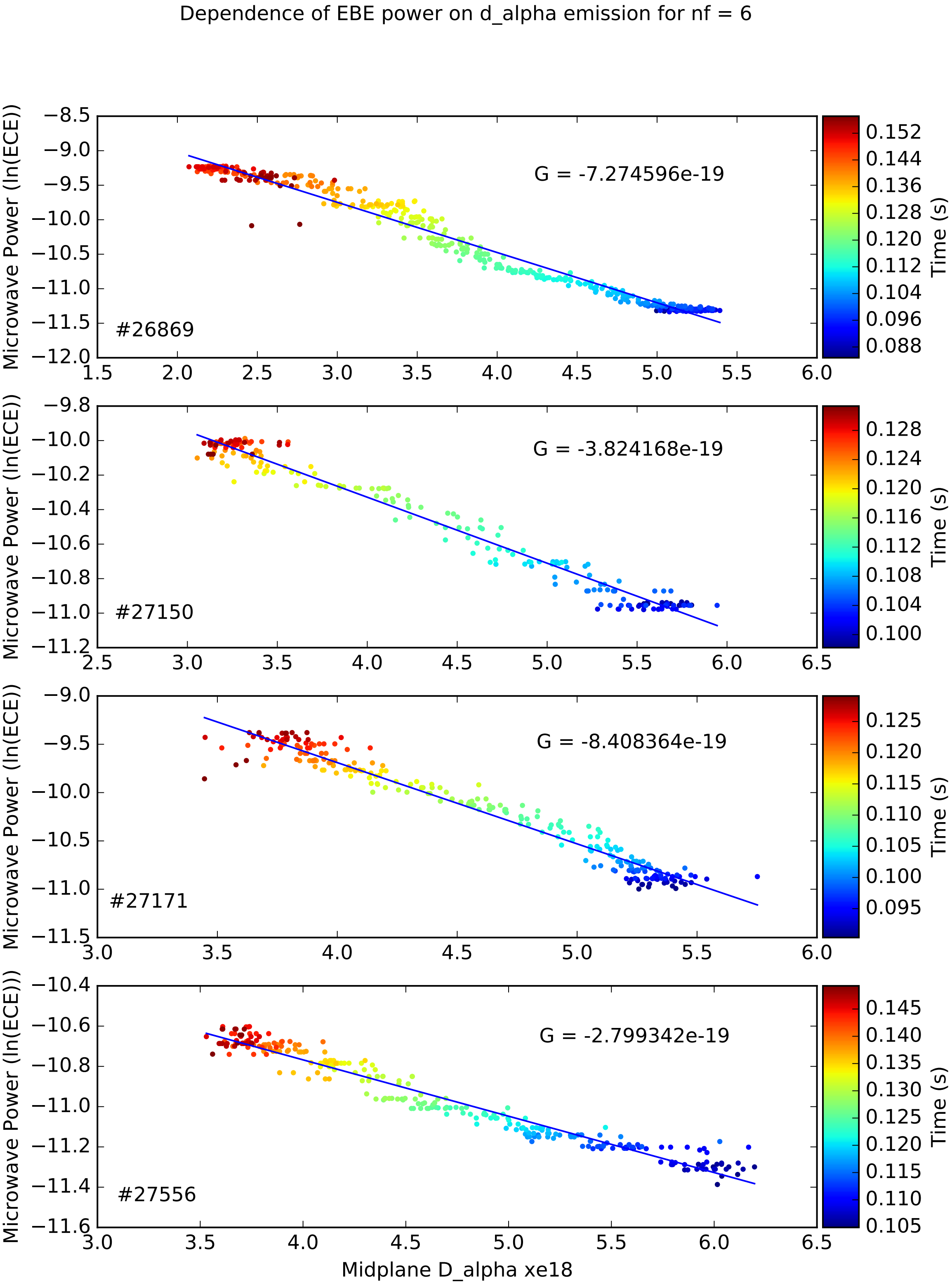


Figure 3.9: Plots of ECE power in antenna 0 vs. D_α emission for shots 26869, 27150, 27171 and 27566 for frequency channel six (15 GHz). The gradient G is the exponent defined in equation 3.7.1.

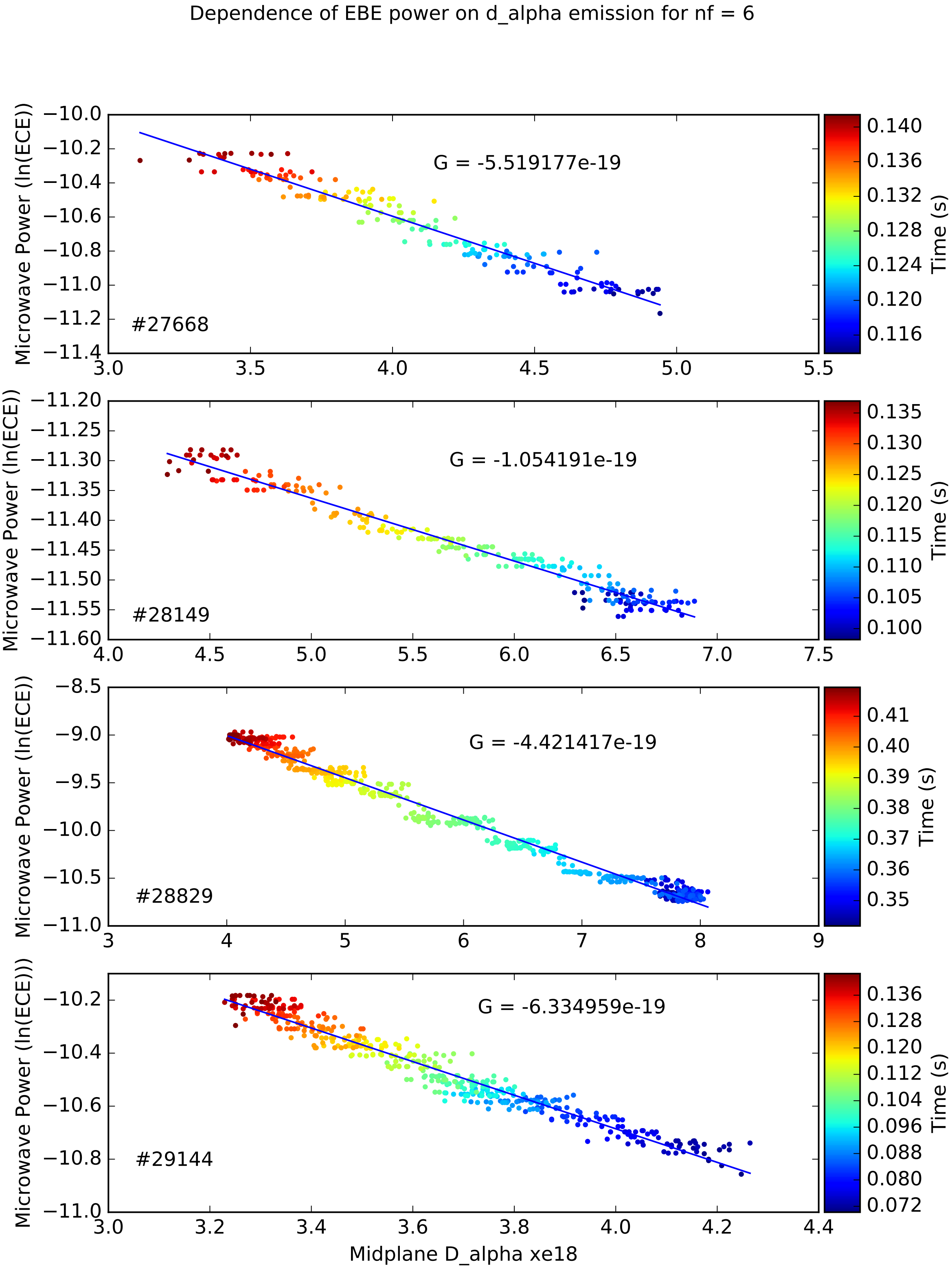


Figure 3.10: Plots of ECE power in antenna 0 vs. D_α emission for shots 27668, 28149, 28829 and 29144 for frequency channel six.

The SAMI frequency channel used in the above plots is frequency channel six, $nf = 6$, or $f_{SAMI} = 15$GHz. We can use the plasma dispersion relation to determine the electron density of the plasma at the mode conversion location:

$$f_{pe} = 8980\sqrt{n_e} \tag{3.7.2}$$

where n_e is the density per cubic centimetre cm^{-3}. Using $f_{pe} = f_{SAMI}$ gives a electron density of $n_e = 2.790$ x$10^{18} m^{-3}$.

Looking at the Thomson scattering data [126] for the relevant time intervals gives radial electron density and temperature profiles for each time step in the interval. Using the density at the MC layer, the radial location of the MC layer for each time step can be inferred and then the electron temperature at this location so that temporal plots of L_{MC} (density divided by density gradient) and T_e can be obtained.

For illustration, Fig. 3.11 and Fig. 3.12 show the radial density and temperature profiles at the first time step in the interval where the relationship between ECE power and D$_\alpha$ emission is observed. The same eight shots discussed in section 3.7.1 are shown with the density profile on the left and the temperature profile on the right. Using the density at the MC layer, it is shown how the position of the MC layer is obtained. This radial location is then used to deduce the temperature.

Carrying this analysis out for each time step in the interval gives temporal profiles of the ECE power, D$_\alpha$, radial location of MC window, density gradient at the MC layer, temperature at the MC layer and the density scale length at the MC layer. This is illustrated in each of the eight shots in Fig. 3.13 and Fig. 3.14 here and in Figs. A.1 - A.6 in Appendix A.6.

These shots show similar behaviour, with all shots exhibiting the increase in ECE as the D_α decreases. Most of the shots have increasing major radius R so the plasma is moving out and expanding over the time window of interest. Most of the shots show the temperature T_e increasing over the time window, as expected, the exception being shot 28149. However, the density gradient, and therefore the scale length L_{MC}, does not decrease as expected and the signal is rather noisy. This is due to the Thomson scattering data often not being stable enough in the edge to calculate the density gradient accurately there.

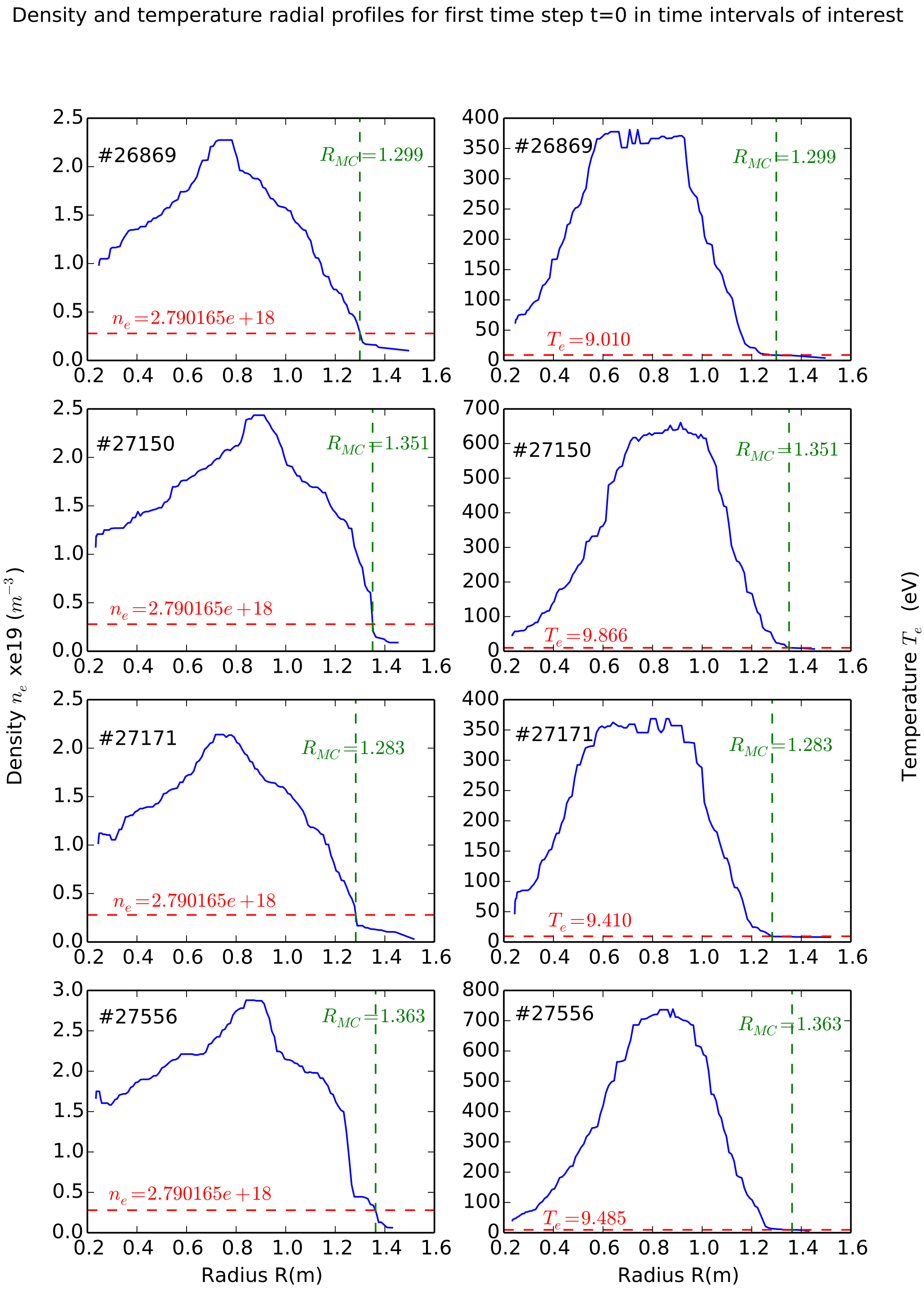


Figure 3.11: Radial density and temperature profiles for shots 26869, 27150, 27171 and 27566 for frequency channel six at t = t_0.

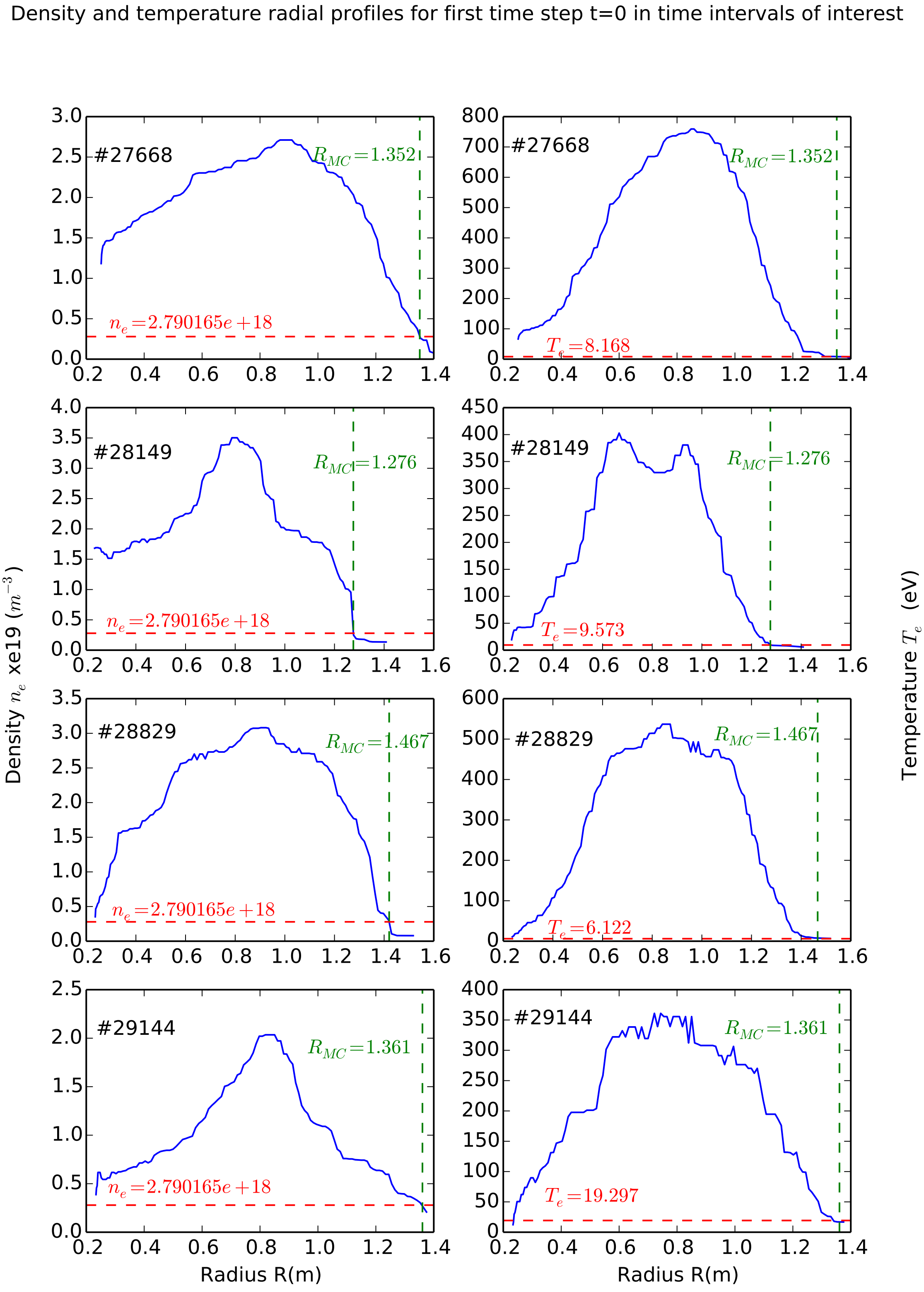


Figure 3.12: Radial density and temperature profiles for shots 27668, 28149, 28829 and 29144 for frequency channel six at t = t_0.

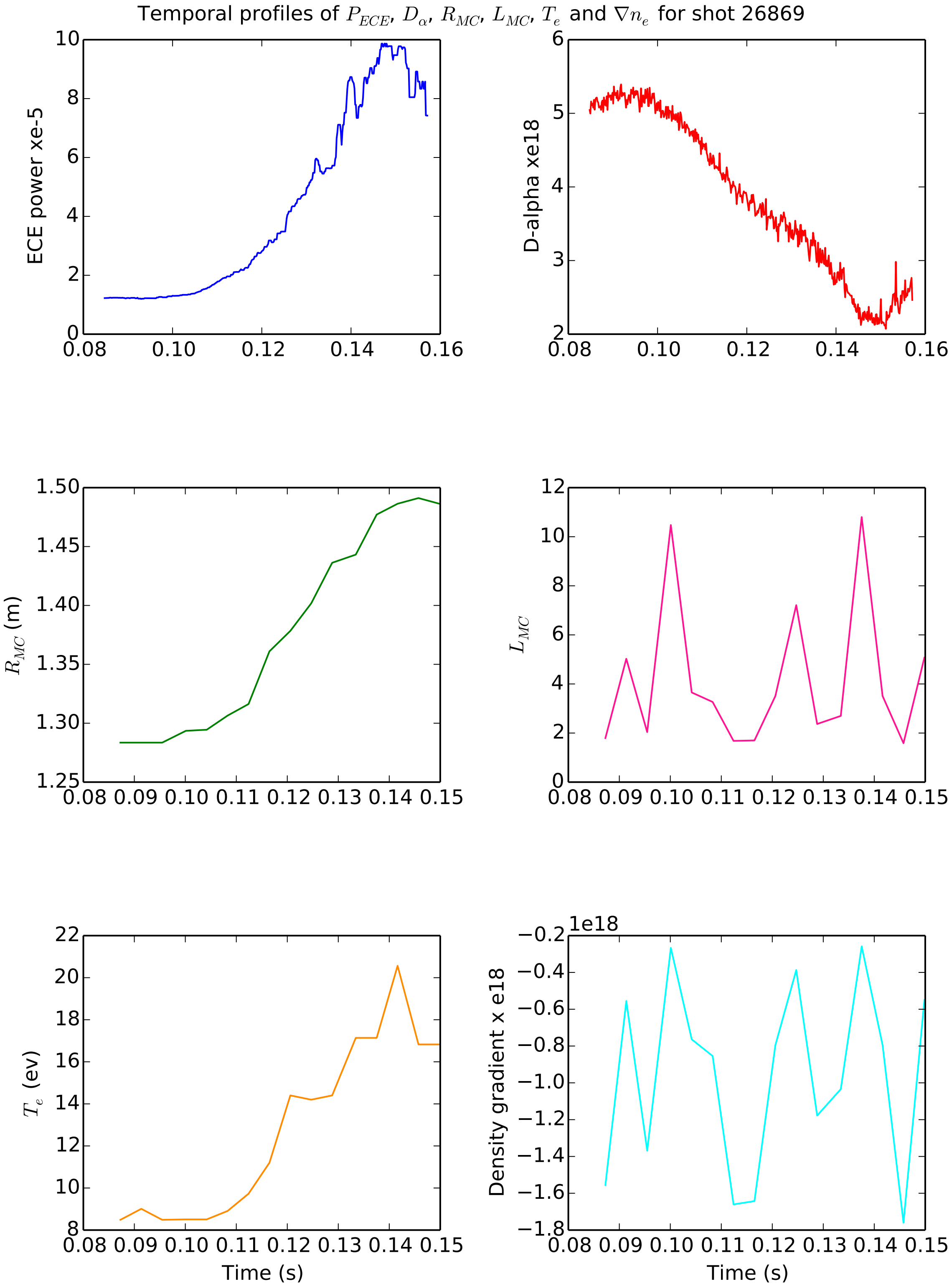


Figure 3.13: Temporal profiles of ECE power (top left), D_{α} emission (top right), radial location of MC window (middle left), density scale length (middle right), electron temperature at MC window (bottom left) and density gradient (bottom right) for shot 26869.

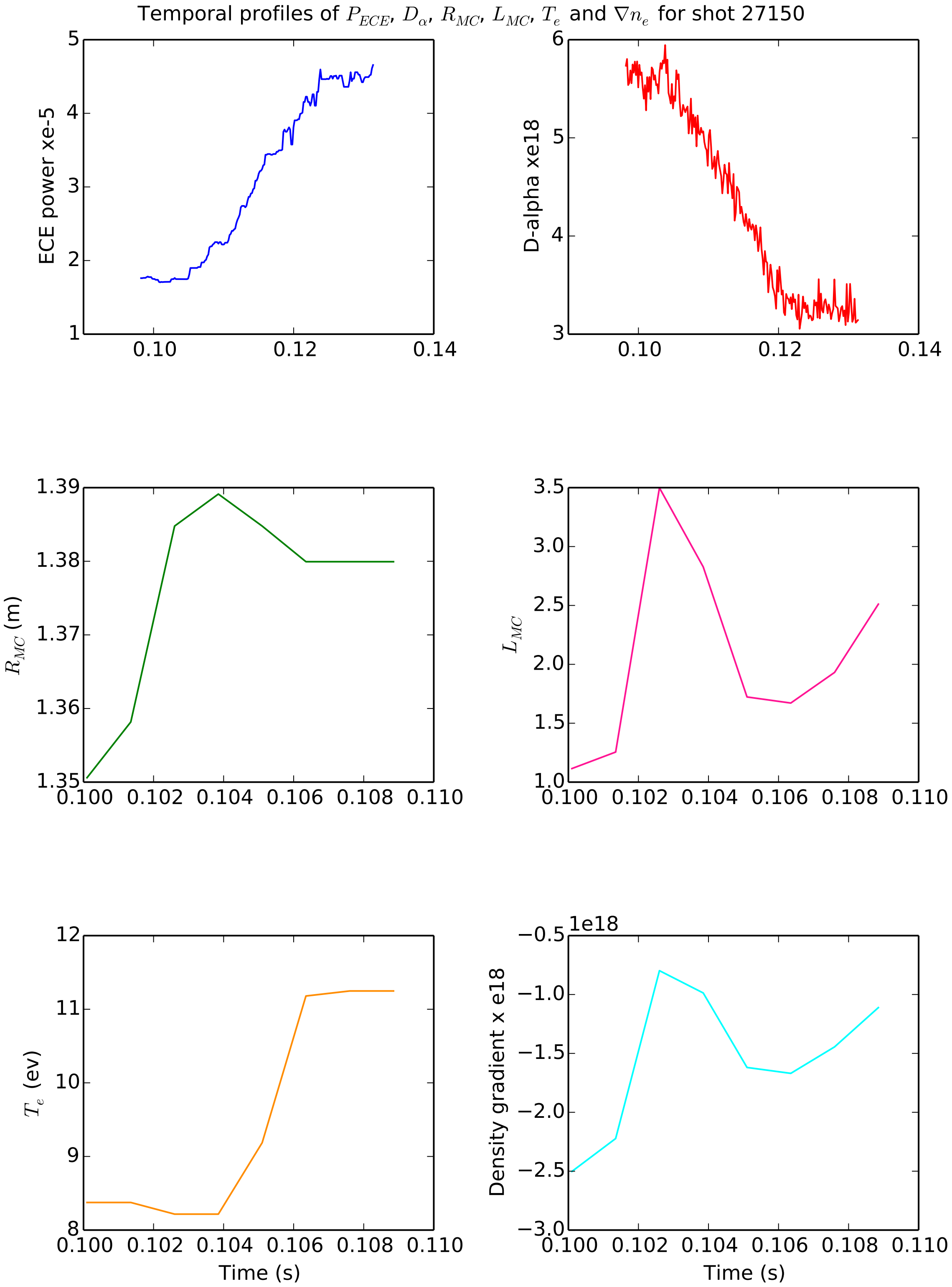


Figure 3.14: Temporal profiles of ECE power, D$_\alpha$ emission, radial location of MC window, density scale length, electron temperature at MC window and density gradient for shot 27150.

The Thomson scattering density, temperature and radial data can be time averaged over the window of interest to improve the signal-to-noise ratio. Then the scale length, density gradient and temperature at the MC location can be calculated. Doing this for each shot that is: *(i)* guaranteed to have the reflectometer off, *(ii)* be a double null shot and *(iii)* be a H-mode shot, results in a dataset of 66 shots where the dependent variable is $\frac{d(ln(P_{ECE}))}{d(D_\alpha)}$ and the independent variables are L_{MC}, T_e and ∇n_e for each shot. A fitting function can then be applied to the data to decide if the data is a good fit to a particular model.

3.7.3 Plasma parameters effecting D$_\alpha$ emission and potential models

The D$_\alpha$ signal used is a line integrated measurement which depends on the density, temperature and scrape off layer width (scale length). Naively, *(i)* $D_\alpha \sim L_{MC} n_e T^{-\frac{1}{3}}$. However D$_\alpha$ light emission in C-Mod and NSTX [127] implies there could be two other dependencies for the D_α emission to consider *(ii)* $D_\alpha \sim n^{0.6} T^{0.5}$ and *(iii)* $D_\alpha \sim n^{0.7} T^{0.5}$.

If the narrowing of the density layer is causing enhanced tunnelling through the MC layer, then we would expect a relationship $P_{ECE} \sim L_{MC} D_\alpha$ as integrating the MC formula gives $P_{ECE} \sim L_{MC}$. Therefore a linear relationship between P_{ECE} and D_α is expected. As there is a log-linear relationship observed, it can be assumed that the narrowing of the density layer is not responsible for the observation of increased ECE power.

If damping is the case then $P_{ECE} \sim e^{-L_{MC} v_{col}}$ where $v_{col} \sim n_e T^{-\frac{3}{2}} \implies P_{ECE} \sim e^{-L_{MC} n_e T^{-\frac{3}{2}}}$. Therefore for each of the three cases *(i), (ii) and (iii)* we have:

$$P_{ECE} \sim e^{-D_\alpha T^{-\frac{7}{6}}} \implies \frac{d(ln(P_{ECE}))}{d(D_\alpha)} \sim T^{-\frac{7}{6}} \qquad (i) \qquad (3.7.3)$$

$$P_{ECE} \sim e^{-L_{MC} D_\alpha n_e^{0.4} T^{-2}} \implies \frac{d(ln(P_{ECE}))}{d(D_\alpha)} \sim L_{MC} n_e^{0.4} T^{-2} \qquad (ii) \qquad (3.7.4)$$

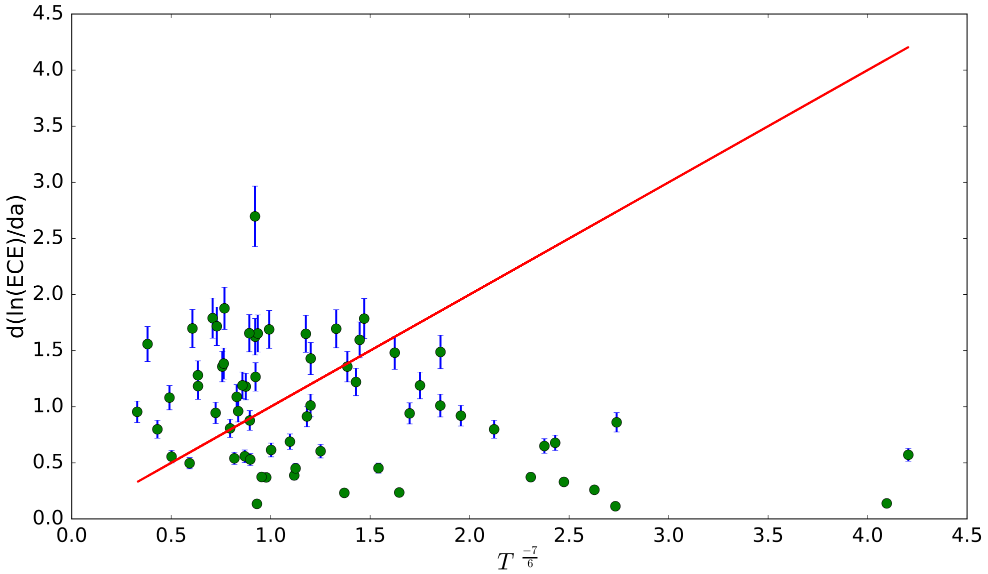


Figure 3.15: Model (i). Green points are the observed data, representing each shot. The red line is the model prediction. Error bars are based on the error associated with the Thomson scattering diagnostic.

$$P_{ECE} \sim e^{-L_{MC}D_\alpha n_e^{0.3}T^{-2}} \implies \frac{d(ln(P_{ECE}))}{d(D_\alpha)} \sim L_{MC}n_e^{0.3}T^{-2} \qquad (iii) \qquad (3.7.5)$$

Applying these models to our data results in the plots shown in Figs. 3.15 - 3.17. Each green point refers to one shot and the red line is the model prediction. Error bars are included and represent the error associated with the Thomson scattering diagnostic temperature and density measurements and are of the order of 5 - 10% [128]. A worse case scenario of a 10% error in the Thomson scattering data has been used here. Error bars have been estimated in the standard way such that if $R = R(X, Y)$, the uncertainty in measurement X is δX and the uncertainty in measurement Y is δY then the uncertainty in R is:

$$\delta R = \sqrt{\left(\frac{\partial R}{\partial X}\delta X\right)^2 + \left(\frac{\partial R}{\partial Y}\delta Y\right)^2} \qquad (3.7.6)$$

For the purposes of estimating the errors, it is assumed here that the temperature and density measurements are independent. Additionally, it is assumed the error in the density gradient is the same as the error in the density measurement. These assumptions may not be valid generally, but serve to provide an estimate of the

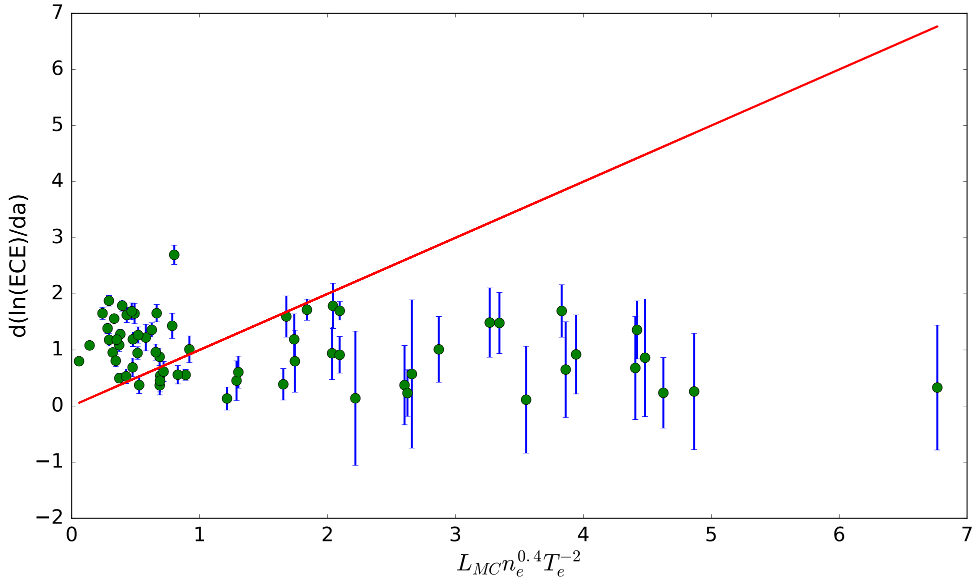


Figure 3.16: Model (ii). Green points are the observed data, representing each shot. The red line is the model prediction. Error bars are based on the error associated with the Thomson scattering diagnostic.

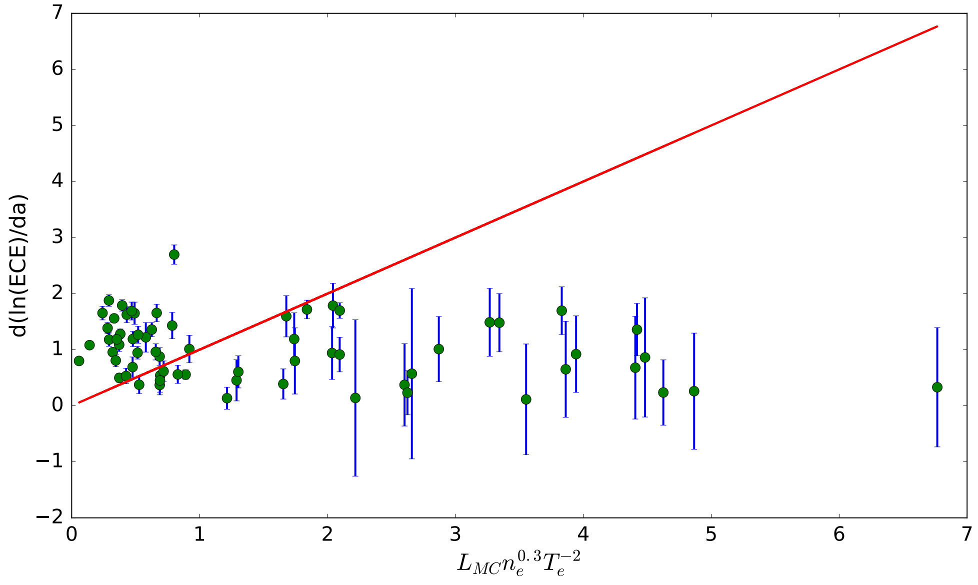


Figure 3.17: Model (iii). Green points are the observed data, representing each shot. The red line is the model prediction. Error bars are based on the error associated with the Thomson scattering diagnostic.

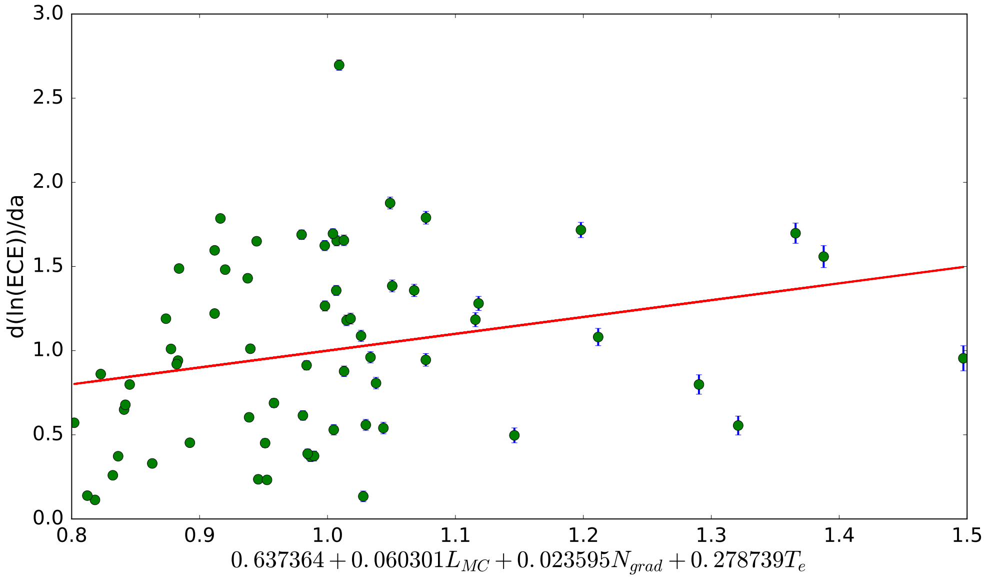


Figure 3.18: Multivariate linear regression for mode conversion parameters L_{MC}, N_{grad} and T_e. Green points are the observed data, representing each shot. The red line is the model prediction. Error bars are based on the error associated with the Thomson scattering diagnostic.

error for these models. We see that none of these models are a good fit for our data. Looking more generally at these variables, a multivariate linear regression has been performed which is shown in Fig. 3.18. None of these models agree well with the observed data however so further plasma quantities need to be considered and added to the model if it is possible they could be effecting the observed ECE power or D_α emission.

3.7.4 Physical processes which may effect the ECE emission

The observed microwave emission could depend on at least three factors: the mode conversion efficiency, the birth intensity of the emission and collisional damping which may occur in the edge plasma. The mode conversion efficiency, or transmission coefficient for O-X-B conversion is given by [129] [130]:

$$T(N_{||}, N_y) = exp\left(-\pi k_0 L_n \sqrt{\frac{Y}{2}}\left(2(1+Y)(N_{||,opt} - N_{||})^2 + N_y^2\right)\right) \tag{3.7.7}$$

where $N_{||}$ is the parallel refractive index, N_y is the y-component of the refractive index, k_0 is the vacuum wavenumber, L_n is the density scale length, Y is the dimensionless electron cyclotron frequency $\frac{\omega_c}{\omega}$ and $N_{||,opt}$ is the optimal $N_{||}$ at the optimal launch angle. Equation 3.7.7 implies that the gradient $G = \frac{d(ln(P_{ECE}))}{d(D_\alpha)}$ will depend on the scale length L_{MC} in some way. However, the above plots imply this is not the case as there doesn't appear to be any dependence on L_{MC}.

Now considering the birth intensity of the wave, Fig. 3.19 and 3.20 show the birth density and birth temperature of the wave and how this evolves over the time windows for each shot. To produce these plots, the MAST data for signals `EFM_PSI(R,Z)`, `EFM_PSI_AXIS`, `EFM_PSI_BOUNDARY` and `EFM_F(PSI)_(C)` were used to calculate the magnetic field. The magnetic field can be derived from the Grad-Shafranov equation so that:

$$|B| = \frac{1}{R}\sqrt{\left(\frac{d\psi}{dz}\right)^2 + \left(-\frac{d\psi}{dR}\right)^2 + F^2} \tag{3.7.8}$$

where $F = RB_\phi$. Taking the midplane (z=0) to avoid the necessity for ray-tracing and using the electron cyclotron resonance with the SAMI frequency channel:

$$\omega_{ce} = \frac{eB}{m} \Longrightarrow B = \frac{2\pi m f_{sami}}{e} \tag{3.7.9}$$

the radial location for that value of the magnetic field can be found, in a similar way to finding the location of the mode conversion window above. The Thomson scattering data for density and temperature was then used to find their values at this birth location. These birth plots indicate that the birth density does not significantly affect the microwave emission as it is roughly constant throughout the time interval. However the birth temperature does increase, sometimes doubles, over the time window of interest so could be an additional variable that influences the observed microwave intensity. This agrees with the ECE occurring at long wavelengths where the Rayleigh-Jeans law for optically thick resonance applies [131], giving the intensity of emission as a function of temperature:

$$I_n(\omega) = \frac{\omega^2 T_e(R)}{8\pi^3 c^2} \tag{3.7.10}$$

Table 3.6: Chi-squared statistic for the different models

Model	**Chi-Squared Statistic**
B $\sim$ T$^{-\frac{7}{6}}$	48.70
B $\sim$ L$_{MC}$ n$_e^{0.4}$ T^{-2}	125.42
B $\sim$ L$_{MC}$ n$_e^{0.3}$ T^{-2}	125.42
B $\sim$ 0.63 + 0.06L$_{MC}$ + 0.02N$_{grad}$ + 0.28T$_e$	18.15
B $\sim$ -0.30 + 0.04n$_{birth}$ - 0.10T$_{birth}$ + 1.36R$_{birth}$	17.65
B $\sim$ -1.15 + 0.12L$_{MC}$ + 0.26T$_e$ + 0.36R$_{edge}$ + 0.15n$_{birth}$ - 0.12T$_{birth}$ + 1.37R$_{birth}$	16.18

It is therefore worth considering the birth temperature as a possible parameter which affects the variable $\frac{d(ln(P_{ECE}))}{d(D_\alpha)}$. Performing another multivariate linear regression, adding in the birth parameters as variables results in Fig. 3.21. And including both the birth parameters and the mode conversion parameters results in Fig. 3.22. Error bars are calculated as above, including the spatial resolution of the diagnostic of 1cm [126] [128] as an estimate of the error on the radial measurements. Neither of these fits is very promising so it is likely there is still something missing from the model.

3.7.5 Comparison of the models

As has already been seen in Figs. 3.15 - 3.18, Fig. 3.21 and Fig. 3.22, the values for the variable G= $\frac{d(ln(P_{ECE}))}{d(D_\alpha)}$ predicted by the different models do not agree well with the observed data. An attempt to quantify how well each model fits the data has been made in terms of the chi-squared statistic for each model. This is shown in Table 3.6.

The chi-squared statistic measures the goodness of fit between the model and the data and values closer to one imply the model is a better fit to the data and accounts for the variance in the measurement errors in the Thomson scattering data. Chi-squared statistics much greater than one, which is the case in all these models, indicates a poor model fit. The smallest chi-squared error is for the model that

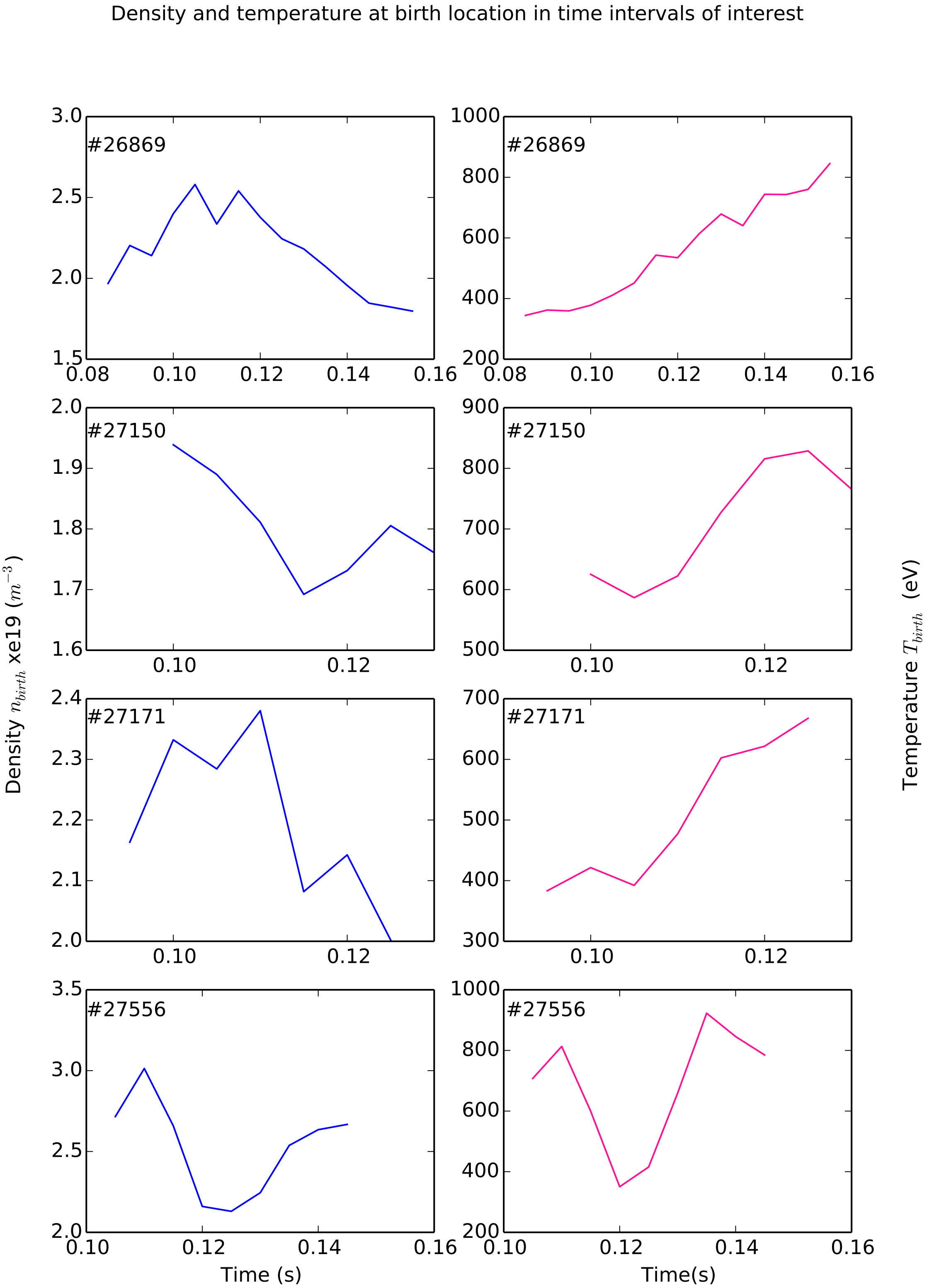


Figure 3.19: Temporal profiles of birth density (left) and birth temperature (right) of microwave emission for shots 26869, 27150, 27171 and 27566.

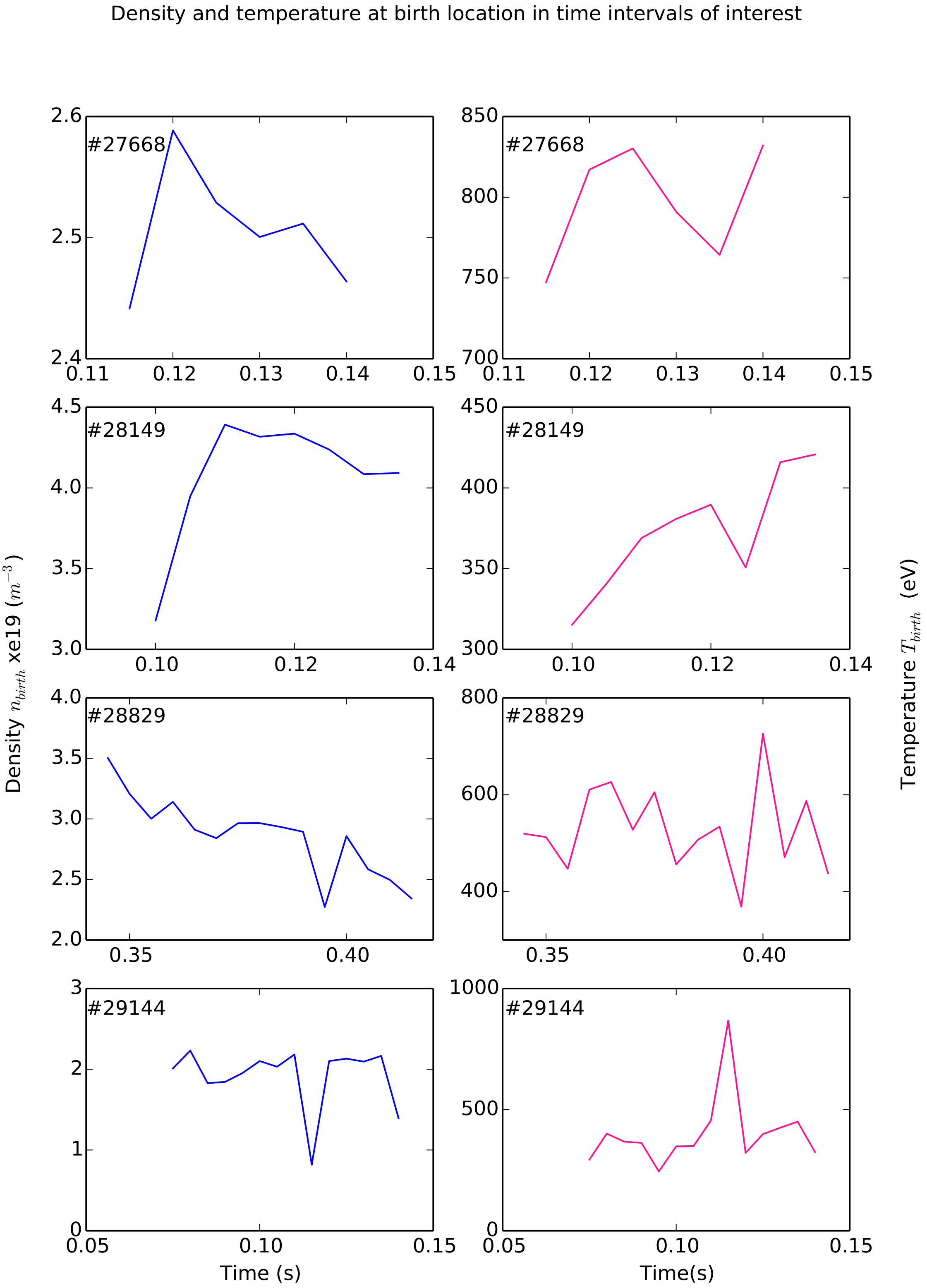


Figure 3.20: Temporal profiles of birth density (left) and birth temperature (right) of microwave emission for shots 27668, 28149, 28829, 29144.

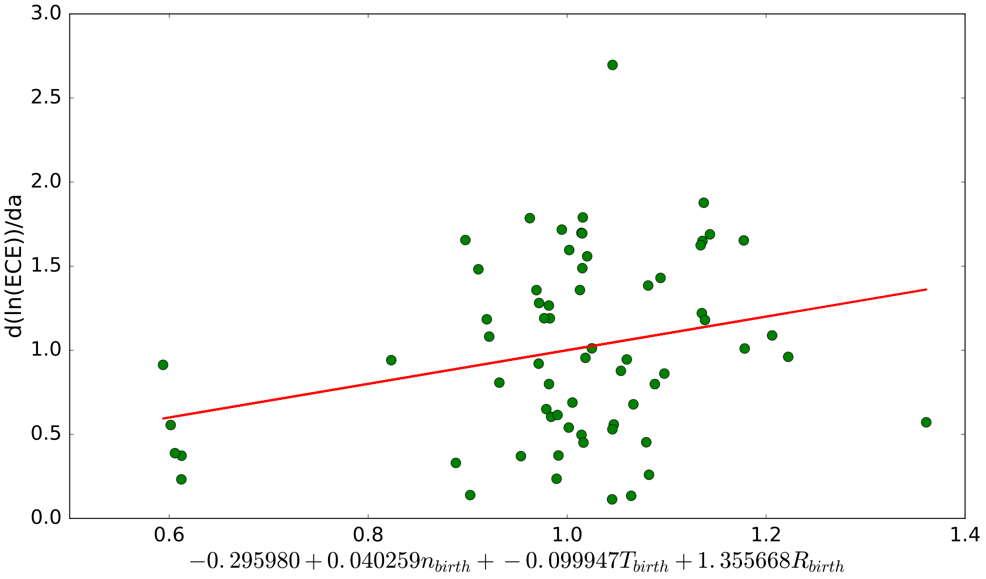


Figure 3.21: Multivariate linear regression with birth parameters only. The red line is the model prediction, green points are the observed data and the error bars are small enough to not show here.

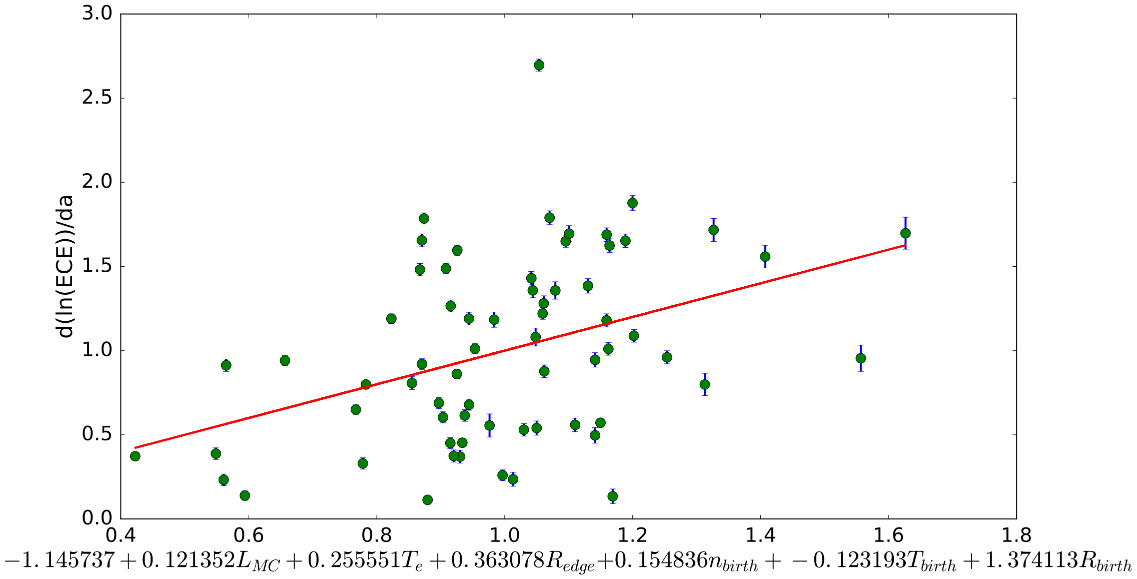


Figure 3.22: Multivariate linear regression with both mode conversion parameters and birth parameters. The red line is the model prediction, green points are the observed data and the error bars are shown in blue and based on errors in the Thomson scattering data.

includes plasma parameters that effect both mode conversion processes and birth intensity of the ECE emission, as expected. Further additions to the model could involve plasma parameters affected by the collisionality of the plasma such as the ion temperature and density, T_i and n_i, as these are likely to effect the ECE emission and also the D$_\alpha$ emission.

3.8 Final words on SAMI data processing code and data analysis

The accelerated GPU data processing code has provided a dramatically reduced runtime to process the raw data from the SAMI diagnostic such that in the future inter-shot processing will be able to be performed. Additionally, the GPU code has enabled the processing of all of the stored raw data from previous MAST campaigns leading to the consistent observation across many SAMI shots of a log-linear relationship between ECE power and D$_\alpha$ emission. A few simple heuristic model ideas were explored to explain this observation based on some of the underlying physics which may be occurring. Unfortunately it has not been possible to quantify in detail the observed relationship and relate it to any physical plasma parameters or processes which are occurring. Further SAMI data from future campaigns, such as the upgraded spherical tokamaks MAST-U and NSTX-U, should allow a wider range of plasma conditions and shot parameters to be explored which may reveal new physical insight.

Chapter 4

A GPU based Bateman solver for nuclear burn up calculations

In this chapter a GPU-based solver for the Bateman equations [132] is introduced which models the variation in nuclide number densities over time when irradiated by neutrons. The underlying physical model is introduced and each part of the simulation discussed in terms of its potential for acceleration. Results are compared against the commonly used tool FISPACT-II [133] and issues with differing numerical methods are discussed.

For a reactor model, the reactor is spatially divided into multiple cells or regions. Currently FISPACT-II calculates the reaction rates and solves the Bateman equations for each reactor region sequentially, cell by cell. It would be advantageous to calculate the reaction rates and solve the Bateman equations for all cells simultaneously. GPU technology provides this capability. The Bateman solver is used within the neutronics community for magnetic confinement fusion, inertial confinement fusion and also fission. A greater understanding of the neutron physics involved in fusion would aid the development of reactor blanket technologies to provide shielding from harmful neutron radiation as well as lead to efficient absorption of the 14.1 MeV neutrons produced, the conversion of this energy, and tritium production to ensure reactor tritium self-sufficiency [134].

The chapter is organised as follows. In section 4.1 the Bateman equations are introduced and possible numerical methods for solution, including time-stepping and matrix exponential methods, on a GPU are discussed. Section 4.2 discusses the FISPACT-II code and algorithm used for the linear equation solver. Section 4.3, 4.4 and 4.5 discuss each part of the simulation in detail; *(i)* the nuclear data cross-section collapse on the GPU, *(ii)* the activation matrix creation and *(iii)* the solution of the system of linear equations on the GPU. Within section 4.5 different numerical methods for solving the system of linear equations by calculating the matrix exponential are considered and the decision to use the Chebyshev rational approximation method (CRAM) is justified. In section 4.6 a FISPACT simulation is considered and the outputs of FISPACT-II and the GPU code are compared and in section 4.7 the acceleration the GPU code provides is presented. In section 4.8 the absolute and relative tolerance settings that FISPACT uses in the solver for the system of linear equations is discussed and it is shown that setting the tolerances stricter improves the agreement between the output of the GPU code and FISPACT.

4.1 Bateman equations

The Bateman equations [132] are used to model the evolution of nuclide number densities as a function of time when they are subjected to irradiation by energetic particles. When particles such as neutrons, protons and deuterons interact with a material, several possible reactions can occur. Depending on the nuclides which are being irradiated, the nuclide may absorb the incoming particle, collide with the incoming particle causing scattering or it may split up into two smaller nuclides as a result of fission. An activation code models these chains of reactions with respect to time by utilizing nuclear databases that define the cross-sections and decay constants for every nuclide. An activation or burn-up code therefore solves the Bateman equations:

$$\frac{dN_i}{dt} = -(\lambda_i + \sigma_i\phi)N_i + \sum_j(\lambda_{ji} + \sigma_{ji}\phi)N_j + S_i\sum_k N_k\sigma_i\phi \tag{4.1.1}$$

where N_i is the number density of nuclide i at time t, λ_i is the decay constant of nuclide i, λ_{ji} is the decay constant of nuclide j producing nuclide i, σ_i is the absorption cross-section of nuclide i, σ_{ji} is the reaction cross-section for nuclide j

producing nuclide i, ϕ is the neutron flux density and S_i is a fission source.

The Bateman equations can also be described in matrix form as:

$$\dot{\mathbf{N}} = \mathbf{A}\mathbf{N} \qquad \mathbf{N}(0) = \mathbf{N}_0 \tag{4.1.2}$$

where $\mathbf{A}$ is known as the activation matrix and $\mathbf{N}$ is the vector of nuclide densities. The solution of equation 4.1.2 which is a first-order linear ordinary differential equation is:

$$\mathbf{N} = e^{\mathbf{A}t}\mathbf{N}_0 \tag{4.1.3}$$

There are essentially two methods to calculate the solution to the Bateman equations:

1. Time-stepping
2. Matrix exponential

and the matrix exponential method was chosen here as this method is better suited to the GPU architecture. Time-stepping methods such as Runge-Kutta are non-trivial to perform on a GPU because they are recursive and the next time step depends on the previous time step. Runge-Kutta methods take multiple steps through time, traversing a sequence of data-dependent stages that are ill-suited to the GPU architecture [135]. However, positive performance gains over CPU only approaches have been obtained by employing techniques to workaround the recursiveness problem. A 2016 study [136] considered three GPU methods for solving the ordinary differential equations in a fission burn up light water nuclear reactor scenario; *(i)* Runge-Kutta-Fhelberg method, *(ii)* Jacobi Collocation method, where the matrix exponential is approximated by Gauss quadrature using Jacobi polynomials and *(iii)* matrix exponential rational approximation using Padé's diagonal method and the scaling and sqauring method. In this fission burn up example, the Runge-Kutta method has the fastest total runtime for the simulation and the scaling and squaring method has the longest total runtime. All methods agreed with the sequential algorithm with a root mean squared error of 1%. We chose to focus on the matrix exponential method because there are many different ways of computing the matrix exponential [137] which will be discussed in section 4.5 and not all of the methods have been attempted on a GPU before. The justification for the chosen matrix exponential method will also be given in section 4.5.

4.2 FISPACT-II code solver

The full burn-up calculation can be split up into 3 parts; *(i)* the pre-processing or calculation of the reaction rates from the nuclear data, *(ii)* the construction of the activation matrix **A** and *(iii)* the solution of the Bateman equations by the matrix exponential method.

The FISPACT-II code has been extensively used in the neutronics community. A neutron shielding and activation study of MAST-U device [138] used multiple neutron spectra on a 3D mesh modelling the tokamak which were then passed into multiple FISPACT runs to calculate a nuclear inventory for each mesh point. However this study [138] is coarse in mesh size and cooling step size and a more detailed and realistic model, requiring many more runs of FISPACT needs to be carried out. A study of tritium generation in solid-type breeder blankets in DEMO [139] uses FISPACT and it was shown a fine spatial resolution is needed to accurately predict the tritium inventory. Whilst suitable for this study [139], for a more complex and detailed blanket design it is suggested a mesh approach to burn-up could be advantageous. In these simulations mesh cells are treated independently and there is no interaction between cells. A GPU-accelerated version of FISPACT would aid both of these examples by being able to perform many FISPACT runs for many mesh cells simultaneously, rather than sequentially, and therefore a finer mesh can be used for a more detailed and reliable simulation without becoming too computationally expensive.

The FISPACT-II code extracts the nuclear data, in this case the energy dependent cross-sections, from the TENDL2015 nuclear data library [140] [141] which follows the ENDF-6 nuclear data format [142]. The ENDF-6 format provides representations for neutron cross-sections and distributions, photon production from neutron reactions, charged particle production from neutron reactions, photo-atomic interaction data, thermal neutron scattering data, and radionuclide production and decay data (including fission products). The energy dependent cross-sections are collapsed with the irradiated projectile flux, to provide an effective cross-section for each reaction. This is essentially a matrix-vector product. The rate equations are constructed using the collapsed cross-sections and decay constants for the coefficients, and solved to determine the time evolution of the inventory in response to different irradiation scenarios. The assumption FISPACT uses is that the reaction rates and decay constants are independent of the current inventory and are determined solely by the physical nuclide properties [133]. Additionally, the projectile flux is assumed to be constant throughout the simulation and not modified by the presence of the target material. This means the rate equations are linear. The system is highly sparse as the activation matrix **A** is sparse with approximately 3% non-zero entries. There are a few nuclides in the nuclear data library that have large decay rates and this ensures that all practical FISPACT-II calculations with the full inventory for many applications are always stiff [133], meaning not all numerical methods that could be used to solve the Bateman equations are stable (unless the step size is prohibitively small so that the simulation runtime is long) and the solver must be chosen with care to ensure accuracy.

The core engine of the FISPACT-II linear, sparse and stiff-ODE solver is the LSODES package [143] which is treated as a black box by FISPACT and no major modifications have been made to the package. LSODES in FISPACT uses the backward differentiation formula method, specifically Gear's method [144], which is a linear multi-step method based on the implicit Adams-Moulton method [145] [146]. LSODES is a successor to Gear's code DIFSUB [147].

As the LSODES solver proceeds, the code automatically adjusts the method order (and the step size) for optimal efficiency while satisfying prescribed accuracy requirements. The accuracy of the calculations is controlled by refining the internal time steps to satisfy a criterion placed on the estimate of the error. The acceptance criterion is based on the sum of relative (rtol) and absolute (atol) tolerances which are user defined and specified by the TOLERANCE keyword in the FISPACT simulation [133]. The error associated with dominant nuclides in a FISPACT-II calculation is determined by the chosen rtol parameter, while for the minor nuclides the tolerance is relaxed by the addition of the atol parameter. This avoids the problems that would occur for a pure relative error estimate in the case of zero or very small nuclide inventories.

LSODES computes a vector **w** from the solution vector **y**:

$$w_i = rtol * y_i + atol \tag{4.2.4}$$

Estimates of the error, e_i, are produced separately for each component of the solution vector and combined with the weights w_i, into a single measure of the error using a root-mean-square:

$$D = \left(\frac{1}{L} \sum_{i=1}^{L} \frac{e_i}{w_i} \right)^{\frac{1}{2}} \tag{4.2.5}$$

where L is the length of the vector **y**. D is used as a single measure of acceptability of the solution; if D >1 then LSODES refines its internal time steps until a satisfactory D is obtained.

The source code for FISPACT-II was not available to study and use in the development of the GPU code which was therefore written from scratch. The executable FISPACT-II code was used to benchmark the simulations and compare output values for the nuclide density vectors with the same simulation performed on the GPU.

4.3 Cross-section collapse

The first step in the burn up simulation is to collapse the cross-sections for each reaction. Each cell in a reactor model which has been spatially divided into multiple cells has S isotopes and within those S isotopes there will be R types of reaction. The calculation of reaction rates is done by collapsing the cross-sections:

$$\langle \sigma_r \phi \rangle = \frac{\sum\limits_{g=0}^{G} \sigma_{rg} \phi_g}{\sum\limits_{g=0}^{G} \phi_g} \tag{4.3.1}$$

where σ_{rg} is the cross-section for reaction r in energy group g and ϕ_g is the neutron flux in energy group g. For computational efficiency the neutron flux is normalised so that $\sum\limits_{g=0}^{G} \phi_g = 1$. The number of energy groups is G=709.

As already stated, presently FISPACT-II calculates the reaction rates one cell at a time and assumes cells don't interact. To calculate the reaction rate for all cells simultaneously the calculation of the reaction rates would become a matrix multiplication:

$$\mathbf{\Gamma} = \mathbf{\Sigma}\mathbf{\Phi} \tag{4.3.2}$$

where $\mathbf{\Sigma}$ is an NxK cross-section matrix containing N nuclear reactions each having K energy regions, $\mathbf{\Phi}$ is a KxM neutron flux matrix containing the neutron flux at M cells for K energy regions and $\mathbf{\Gamma}$ is an NxM reaction rate matrix for M burn zones each having N reactions.

An accelerated matrix multiplication is easy to achieve on a GPU using cuBLAS [148] (see Appendix B.1). Previously a typical burn-up calculation would involve 50-100 isotopes, however the full nuclear database contains $S = 3785$ isotopes so the calculation is extended to include these isotopes. Therefore the dimensions of the calculation are $N = \sum\limits_{S} R_S$, $K = 709$ and M as large as possible to represent the different burn zones, or cells, in the reactor. M is limited only by the available GPU memory. Once the memory required for the problem doesn't fit inside GPU memory there is a performance hit due to having to do the multiplication in batches with CUDA streams as discussed in Chapter 3.

For the S isotopes each having a number of reactions R, using the TENDL2015 nuclear data library [140] [141], $N = 218835$. The dimensions of the matrix multiplication problem become $\mathbf{\Gamma} = [218835, M], \mathbf{\Sigma} = [218835, 709]$ and $\mathbf{\Phi} = [709, M]$.

4.3.1 Accuracy of cross-section collapse

The FNS_INCONEL example is considered which is a "getting started" example as part of the FISPACT installation simulating the five minute irradiation of an Inconel-600 sample and is known to fit the experimental results from such an irradiation. Inconel is an alloy consisting of Nickel, Chromium, Iron and Manganese isotopes that is well-suited to the extreme temperature and pressure environment present in a tokamak. Inconel is the material the JET vessel is made out of and has been in place for the duration of JET's operating life [149]. Recently the ITER-like wall of tungsten and beryllium is the plasma facing component in JET and for the ITER vessel there will be no Inconel as the neutron irradiation expected in ITER will activate several of the components of Inconel. However, this is a good example to consider as it is known the FISPACT simulation matches experimental results well so that the GPU output can be compared objectively to the FISPACT output.

Looking at just one cell for now, the collapsed cross-sections for the FNS_INCONEL simulation from the FISPACT output can be compared with the GPU output to ensure accuracy at this stage. Fig. 4.1 shows both the collapsed cross-sections returned by FISPACT (red-crosses) and the collapsed cross-sections returned by the GPU code (blue circles) for all 106236 reactions that FISPACT considers, each with a unique reaction ID based on the isotope MAT number and the reaction MT number. The MAT and MT number describe the nuclear data based on the ENDF-6 format. Each isotope has its own unique MAT number and each reaction is specified by a unique MT number such that the unique reaction ID = MAT*1000 + MT. Certain reactions are not included in the simulation, those with MT = 18-21, 51-91 and MT > 200, and can be ignored. Additionally, FISPACT only outputs collapsed cross-sections greater than 10^{-12}. Once these considerations have been taken into account the 218835 collapsed cross-sections the GPU calculates are reduced to 106236 and match very well with the FISPACT collapsed cross-sections as shown. To emphasise this point, the relative error between the FISPACT and GPU output for each reaction is shown in Fig. 4.2.

The GPU code is in good agreement with the FISPACT code for the cross-section collapse stage of the simulation. Now let us consider the acceleration provided by the GPU code.

4.3.2 Performance of cross-section collapse

Acceleration results for the cross-section collapse on the GPU using CUBLAS for the simulation FNS_INCONEL are shown in Fig. 4.3. The GPU used is the GeForce

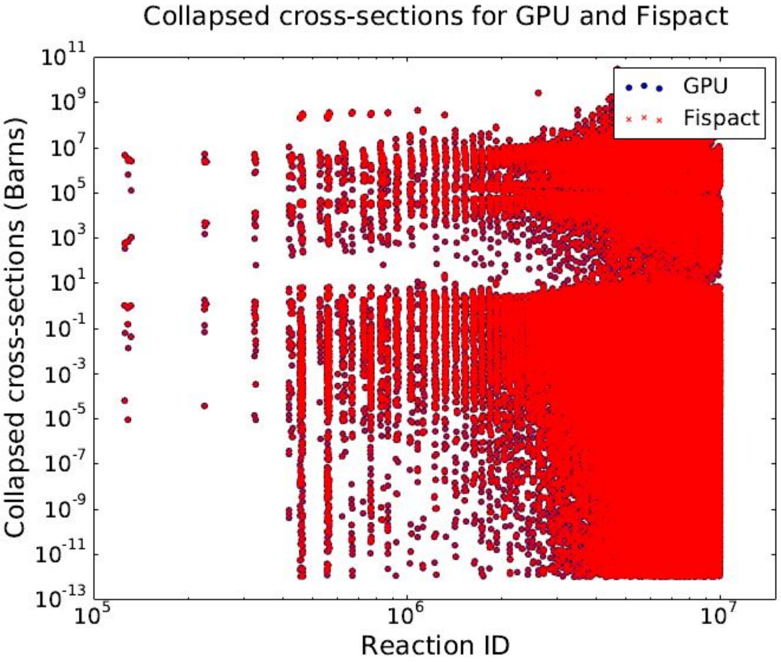


Figure 4.1: Collapsed cross-sections for FISPACT and GPU simulation for each reaction identified by a unique reaction ID.

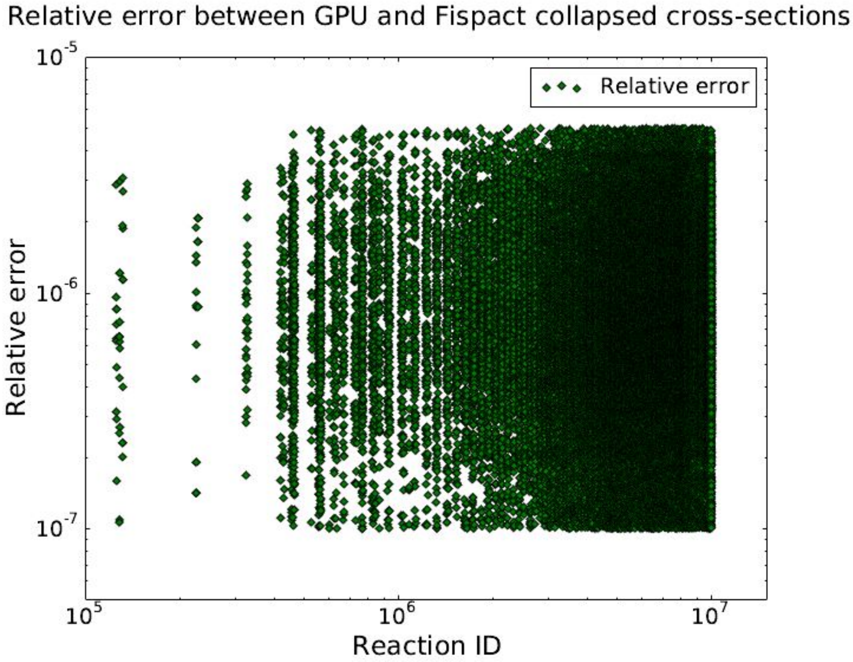


Figure 4.2: Relative error between FISPACT and GPU. Agreement is between $10e^{-5}$ and $10e^{-7}$.

GTX 680 with a very small 2GB GDDR5 memory and FISPACT is run on the Intel i5-5300U CPU. The available GPU memory is so small here that for a problem this size to fit in memory, M can be at most approximately 500. As the aim is to have M as large as possible to model many cells, the matrix $\Sigma = [218835, 709]$ was split into 3 smaller matrices of size [72945, 709] to do the multiplication. This means M can be 3000 before the problem again becomes too large to fit into memory and CUDA streams and concurrency are used, introducing the performance hit as there is extra time involved in organising the flux matrix for efficient processing on the GPU and reorganising the collapsed cross-section output into the correct order. To do the collapse in FISPACT, `fispact collapse` is run which collapses the reaction rates for each isotope with the neutron fluxes for one cell in the reactor using a matrix-vector product. Including all isotopes in the ENDF-6 TENDL2015 library and collapsing with the fluxes has a CPU time of 62.2 seconds as reported by FISPACT. So to do M cells, each with a different neutron flux, the time to do the full collapse is 62.2*M seconds. This is a fair assumption to make as when doing a multi-flux simulation with 41 flux channels the total time taken to perform the collapse is 2447 seconds so we see the time for the multi-flux simulation is approximately 41 times longer.

Memory copy times to and from the GPU are included. The `fispact collapse` command reads the raw nuclear data from the TENDL library which contains many gigabytes of data and is therefore time consuming. Additionally, FISPACT calculates the uncertainties associated with the nuclear data by calculating the covariances where as the GPU code does not. To speed up the collapse for FISPACT and to have a fair comparison between the cross-section collapse in FISPACT and on the GPU, the ENDF libraries can be preprocessed to create a single compressed binary file containing only the cross-section data and none of the variance, resonance or covariance data. This preprocessing has a CPU time of 72.5 seconds and then the collapse can be done in 9.95 seconds CPU time. FISPACT is now approximately six times faster when performing the collapse and only considering the cross-section data. This fairer comparsion between GPU and FISPACT performance is shown in Fig. 4.4 and the GPU still has a huge performance advantage over FISPACT for

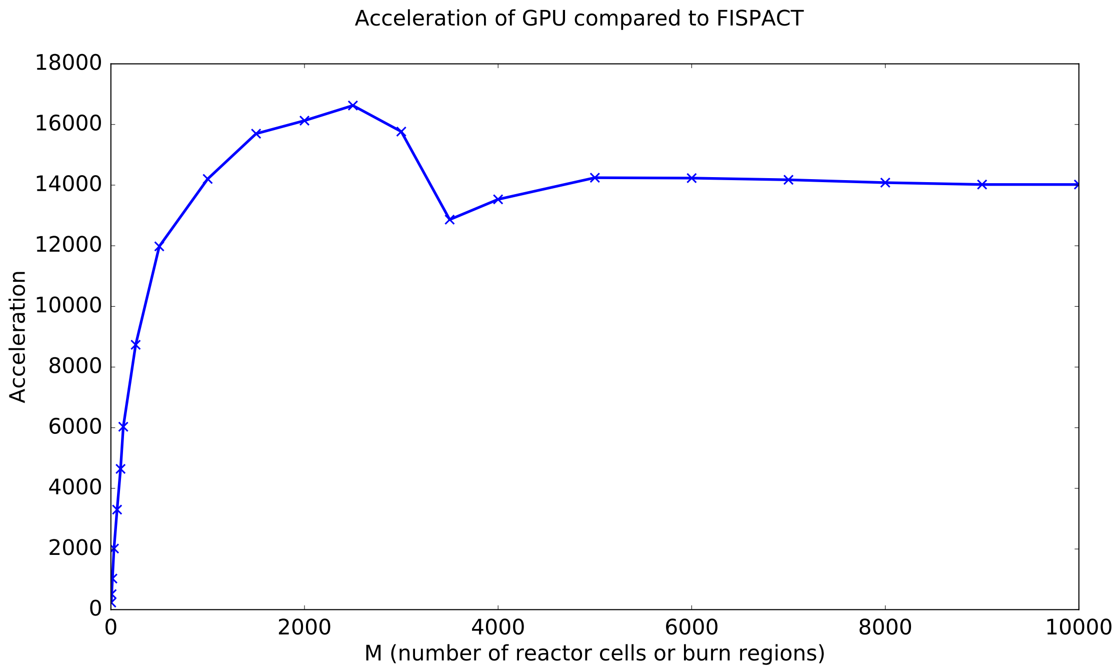


Figure 4.3: Acceleration of CUBLAS cross-section collapse over FISPACT for increasing number of cells M.

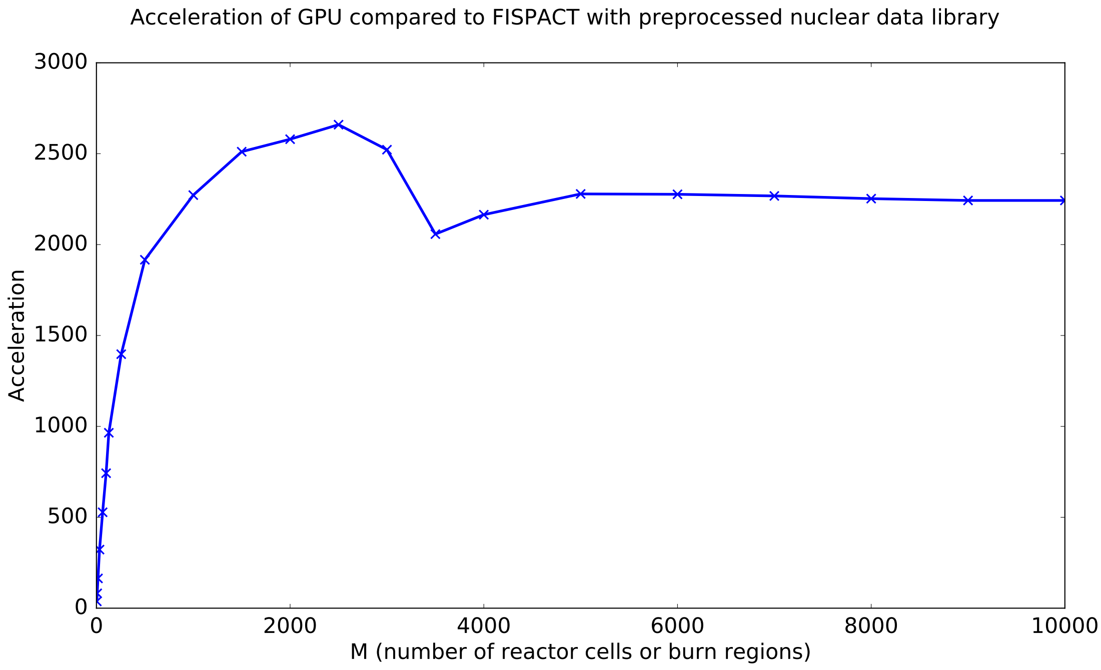


Figure 4.4: Acceleration of CUBLAS cross-section collapse over FISPACT collapse using a preprocessed ENDF library including only the cross-section data.

the cross-section collapse.

Both figures show a decrease in the acceleration of the GPU collapse at $M = 3000$ cells. This is because there is only 2GB memory on the GeForce GTX 680 which can not fit the entire problem of this size so CUDA streams and concurrency must be used as described in Chapter 3 to split the matrix up into chunks. Each chunk is processed in turn, where there is overlap between memory copies and kernel execution on the GPU. Again, memory copy times are included, as is the time taken to rearrange the matrices into an efficient order to be processed by the GPU. This problem can be alleviated or postponed by using a larger GPU, such as the Tesla K40C with 12GB GDDR5.

The acceleration of the cross-section collapse on the GPU is large, even when the ENDF library has been preprocessed and only the cross-section data is considered for the FISPACT collapse. The ability to calculate the collapsed cross-sections for many flux scenarios rapidly is very beneficial to the community, for example being able to perform multi-flux studies for anticipating acceptable levels of neutron flux for DEMO design. FISPACT has been used for a typical DEMO design study to evaluate four different combined cooling and tritium breeding concepts and approximately assess the respective damage doses experienced by the different materials under different neutron flux scenarios [150]. Another study [151], involving tens of thousands of FISPACT inventory simulations, evaluated the activation, transmutation and primary damage response under neutron irradiation with special focus given to the in-vessel conditions predicted for DEMO. Being able to perform a large number of FISPACT simulations in parallel on a GPU is clearly advantageous.

4.4 Creating the activation matrix

The collapsed cross-sections or reaction rates for each cell, $\sigma_r \phi$, are placed into the activation matrices $\mathbf{A}_b$ in the position specified by the reaction for the parent and the daughter and secondary products. Each burn zone b, containing N reaction rates creates one SxS activation matrix $\mathbf{A}_b$. Each reaction rate gets subtracted from the diagonal specified by the parent isotope in the reaction. Additionally, each reaction rate gets added to the location specified by its daughter product and any secondaries. Finally, the decay constants λ_s need to be added to each $\mathbf{A}_b$. The location to add the decay constant to is specified by the type of decay and the branching ratio for that decay. Each λ_s is subtracted from the diagonal location specified by the isotope. Additionally, the decay constant is added to the location specified by the decay, β^-, β^+, α etc. and multiplied by the branching ratio for that decay.

A simple example to illustrate this is one in which there are three nuclides to be considered, ^{3}H, ^{6}Li and ^{9}Be, with the following reactions and collapsed cross-sections:

- ^{3}H (n,total), with collapsed cross-section = 4
- ^{3}H (n,2n), with collapsed cross-section = 4
- ^{6}Li (n,γ), with collapsed cross-section = 6
- ^{6}Li (n,α), with collapsed cross-section = 7
- ^{9}Be (n,total), with collapsed cross-section = 10
- ^{9}Be (n,t), with collapsed cross-section = 8
- ^{9}Be (n,2n), with collapsed cross-section = 2
- $\lambda_{tritium} = 3$ with branching ratio 100%, for the β^- reaction

The resulting activation matrix would be a 9×9 matrix including the isotopes ^{1}H, ^{2}H, ^{3}H, ^{3}He, ^{4}He, ^{6}Li, ^{7}Li, ^{7}Be and ^{9}Be:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & -7 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 7 & -13 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 8 & 0 & -10 \end{bmatrix}$$

To walk through this example:

- ^{3}H (n,total) has no effect on the activation matrix as the change in proton and neutron number, Δp and Δn, are both zero.

- ^{3}H (n,2n) has $\Delta p = 0$ and $\Delta n = -1$ so ^{3}H becomes ^{2}H, the daughter nuclide, and A[2][1] = 4. There are no secondaries. There is the loss term on the diagonal for the parent nuclide ^{3}H so A[2][2] = -4.

- ^{6}Li (n,γ) has $\Delta p = 0$ and $\Delta n = 1$ so the daughter nuclide is ^{7}Li and A[5][6] = 6. There are no secondaries. The loss term on the diagonal for the parent nuclide is A[5][5] = -6.

- ^{6}Li (n,α) has $\Delta p = -2$ and $\Delta n = -1$ so the daughter is ^{3}H and A[5][2] = 7. Secondary nuclide is ^{4}He so A[5][4] = 7. Loss term gets subtracted from diagonal A[5][5] = -13.

- ^{9}Be (n,total) has no effect on **A** as Δp and Δn are both zero.

- ^{9}Be (n,t) has $\Delta p = -1$ and $\Delta n = -1$ so the daughter nuclide is ^{7}Li and A[8][6] = 8. Secondary nuclide is ^{3}H so A[8][2] = 8. The loss term for the parent nuclide is A[8][8] = -8.

- ^{9}Be (n,2n) has $\Delta p = 0$ and $\Delta n = -1$ so the daughter is ^{8}Be and this isotope is not included in the model. There are no secondaries. So there is only the loss term for the parent nuclide subtracted from A[8][8] = -10.

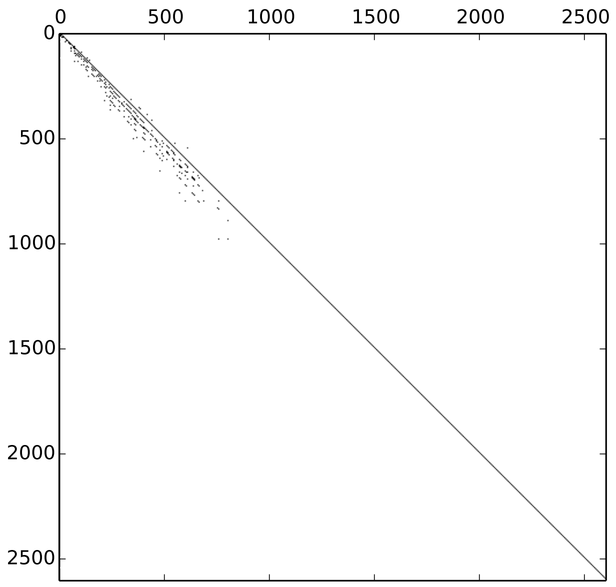


Figure 4.5: Activation matrix of size 2604 × 2604 produced for one burn cell in the FNS_INCONEL example showing the sparsity structure. There are 23856 non-zero entries in total.

- $\lambda_{tritium} = 3$, has $\Delta p = 1$ and $\Delta n = -1$ and this is subtracted from the diagonal for tritium at A[2][2] = -7. It is also added to the location specified by the daughter ^{3}He, A[2][3] = 3

The above activation matrix creation is for the irradiation phase. For the cooling phase, only the decay constants need to be included in the activation matrix.

The resulting activation matrices are very sparse. An example of the sparsity structure for the activation matrix produced in the FNS_INCONEL example we have been considering so far is shown in Fig. 4.5. There are 23856 non-zero entries in this 2604 × 2604 activation matrix so the matrix is only 0.35% filled.

It will be important to exploit this sparsity structure in the next steps of the simulation to save space on the GPU and avoid doing unnecessary computation.

4.5 Calculating the matrix exponential

There are many different methods of calculating the matrix exponential, most of which are outlined in detail in an extremely thorough review of the field [137]. Each method has its advantages and disadvantages. However most of the methods are of dubious numerical quality and suffer from severe round-off errors.

4.5.1 Methods for calculating the matrix exponential

The most obvious way to calculate the matrix exponential is to use a power series or Taylor series and truncate at some point. The matrix exponential is defined in a similar way to the scalar definition:

$$e^{\mathbf{A}} = \sum_{n=0}^{\infty} \frac{\mathbf{A}^n}{n!} = \mathbf{I} + \mathbf{A} + \frac{\mathbf{A}^2}{2!} + \frac{\mathbf{A}^3}{3!} + ... \tag{4.5.1}$$

With this method there is concern where to truncate the power series for efficiency. Additionally, due to round-off error in floating point arithmetic, it is known this method fails to calculate the correct matrix exponential for some matrices.

Another method involves the Padé approximation to $e^{\mathbf{A}}$, defined by [137]:

$$R_{pq}(A) = [D_{pq}(A)]^{-1} N_{pq}(A) \tag{4.5.2}$$

where

$$N_{pq}(A) = \sum_{j=0}^{p} \frac{(p+q+j)!p!}{(p+q)!j!(p-j)!} A^j \qquad D_{pq}(A) = \sum_{j=0}^{q} \frac{(p+q+j)!q!}{(p+q)!j!(q-j)!} (-A)^j \tag{4.5.3}$$

Round-off error also makes the Padé approximation unreliable [137].

The round-off error problem associated with these two methods can be reduced by using the scaling and squaring method:

$$e^{\mathbf{A}} = (e^{\frac{\mathbf{A}}{m}})^m \tag{4.5.4}$$

where m is chosen to be a power of 2 so that $e^{\frac{\mathbf{A}}{m}}$ can be computed reliably by the Padé approximation. The solution $(e^{\frac{\mathbf{A}}{m}})^m$ is then formed by repeated squaring. By restricting $\|A\|/m \leq 1$, $e^{\mathbf{A}/m}$ can be computed accurately with the Padé approximation.

Another method for calculating the matrix exponential involves calculating the eigenvalues and eigenvectors of the matrix and is based on a similarity transformation $\mathbf{A} = \mathbf{SBS}^{-1}$ so that $e^{t\mathbf{A}} = \mathbf{S}e^{t\mathbf{B}}\mathbf{S}^{-1}$ where $\mathbf{S}$ is chosen to be the matrix whose columns are eigenvectors of $\mathbf{A}$. We have the eigenvalue equation $Av_j = \lambda_j v_j$ for $j = 1, ..., n$ and taking $\mathbf{V} = [v_1]...[v_n]$ gives $\mathbf{AV} = \mathbf{VD}$ for $\mathbf{D} = diag(\lambda_1, ..., \lambda_n)$. Then $e^{t\mathbf{A}} = \mathbf{V}e^{t\mathbf{D}}\mathbf{V}^{-1}$ and taking $\mathbf{S} = \mathbf{V}$ we have the desired similarity transformation. Now $e^{t\mathbf{D}} = diag(e^{\lambda_1 t}, ..., e^{\lambda_n t})$ is easy to compute if we have a valid method for calculating the scalar exponential. Difficulty occurs for this method when the matrix $\mathbf{A}$ does not have a linearly independent set of eigenvectors so that the matrix $\mathbf{V}$ is invertible.

Of the methods discussed in this extensive paper [137], it is generally agreed the most stable method is the scaling and squaring method with Padé approximation. However, as discussed in section 4.1 and the 2016 study [136] this method does not perform well on a GPU so is not considered.

4.5.2 Chebyshev rational approximation method

Previous fission burn up studies [152] [153] discuss why the above matrix exponential methods are not good for activation matrices such as those found in burn-up calculations where there are large differences of several orders of magnitude between values in the activation matrix and the activation matrices are very sparse. An alternative method, the Chebyshev rational approximation method (CRAM) is proposed in these studies [152] [153] which outperforms the more conventional matrix exponential methods in terms of computational accuracy and efficiency. CRAM is the best rational approximation on the negative real axis which is where the eigenvalues of the burn up matrix lie. For a rational function $r_{k,k}(z) = p_k(z)/q_k(z)$ with p_k and q_k being polynomials of order k, the CRAM approximation of order k is defined as:

$$r_{k,k}(x) = \alpha_0 + 2Re\left(\sum_{j=1}^{k/2} \frac{\alpha_j}{x - \theta_j}\right) \tag{4.5.5}$$

where α_0 is the limit of the function $r_{k,k}$ at infinity and α_j are the residues at the poles θ_j.

The CRAM method is able to treat both short-lived and long-lived nuclides simultaneously unlike other methods. This method is also mentioned in the paper previously discussed [137]. Consider $max|r_{k,k}(x) - e^{-x}|$ where $r_{k,k}$ is the ratio of two polynomials. The coefficients of the particular $r_{k,k}$ which minimises this maximum can be translated into bounds for $\|r_{k,k}(A) - e^{A}\|$. These coefficients have been determined [154]. CRAM is effective for sparse matrices and matrices with eigenvalues which lie on the negative real axis such as the activation matrices found in burn-up calculations. Given the success of the CRAM method in fission burn up calculations, and the similarities between fission and fusion burn up calculations, we have chosen the CRAM method to accelerate with a GPU. To our knowledge the CRAM method has not been cast onto a GPU before. The GPU implementation is based on MATLAB codes [153] (shown in Fig. 4.6) for the method of calculating the matrix exponential. The GPU implementation is provided in Appendix B.2. The code follows from the rational approximation to equation 4.1.3 and can be written as [153]:

$$\mathbf{n} = \alpha_0 \mathbf{n}_0 + 2Re\left(\sum_{j=1}^{k/2} \alpha_j (\mathbf{A}t - \theta_j \mathbf{I})^{-1} \mathbf{n}_0\right) \tag{4.5.6}$$

where we are using $r_{k,k}(\mathbf{A}t)$ as the rational approximation to $e^{\mathbf{A}t}$ from equation 4.5.5 and multiplying by $\mathbf{n}_0$.

4.6 Accuracy of results for FNS_INCONEL

Considering just one time step in the simulation, an irradiation phase of 5 minutes, with initial conditions listed in Table 4.1, different numerical methods can be explored to emphasise the problems associated with calculating the matrix exponential.

```
function [theta, alpha, alpha_0 = quad_coeffs(k)
    %k = 34 is degree of approximation

    x = pi*(1 : 2 : k - 1)/k;
    theta = k*(0.1309 - 0.1194*x.^2 + 0.2500*x*1i);
    w_j = k*(-2*0.1194*x + 0.2500*1i);
    alpha = 1i*(1/k)*exp(theta).*w_j;
    alpha_0 = 0;
end
```

```
function n = rat_apprx(theta, alpha, alpha_0, A, t, n_0)
    %theta = poles of rational function  r
    %alpha = residues at these poles
    %alpha_0 = limit of r at infinity
    %A is burn up matrix
    %t is time step
    %n_0 is initial composition vector

    s = length(theta);
    A = A*t;
    n = 0*n_0;

    for j = 1 : s
       n = n + (A - theta(j)*eye(size(A))) \ (alpha(j)*n_0);
    end

    n = 2*real(n);
    n = n + alpha_0*n_0;
end
```

Figure 4.6: The MATLAB code for the CRAM method of calculating the matrix exponential and solving the Bateman equation as found in [153].

Table 4.1: Initial nuclide densities

Nuclide	Initial density	Nuclide	Initial density
^{50}Cr	8.03664E+19	^{57}Fe	1.78691E+19
^{52}Cr	1.54979E+21	^{58}Fe	2.37805E+18
^{53}Cr	1.75733E+20	^{58}Ni	5.29597E+21
^{54}Cr	4.37438E+19	^{60}Ni	2.03999E+21
^{55}Mn	4.27506E+19	^{61}Ni	8.86850E+19
^{54}Fe	4.92898E+19	^{62}Ni	2.82703E+20
^{56}Fe	7.73744E+20	^{64}Ni	7.20371E+19

4.6.1 FISPACT vs. MATLAB

The activation matrix produced by the previous step in the simulation can be fed into various MATLAB solvers to explore their numerical convergence. Fig. 4.7 shows the results produced by the FISPACT simulation and the results produced from the MATLAB CRAM method shown in Fig. 4.6. The results agree well, indicating CRAM is indeed a good method to use for calculating the matrix exponential. The MATLAB operation $\mathbf{A}\backslash\mathbf{b}$ involved in the MATLAB CRAM code means to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$. To do this MATLAB analyses the matrix and decides which algorithm path to follow based on the properties of the matrix [155]. All of this is done automatically, without any input from the user. Efficient methods of solving different types of systems of linear equations is the result of years of work by Timothy Davis and others, whose routines have been incorporated into the MATLAB backslash operator [156] [157] [158] [159].

The comparison of FISPACT and the MATLAB routine `expm` [160] also agree well, as shown in Fig. 4.8, indicating that CRAM is as accurate as this function. The scaling and squaring method used in the MATLAB routine `expmdemo1` [161] in Fig. 4.9 also stands up and performs well with good agreement to FISPACT.

However, both the Taylor series method in Fig. 4.10 and the Eigenvalue method in Fig. 4.11, which are the MATLAB routines `expmdemo2` and `expmdemo3` [161] respectively perform poorly. The `expmdemo2` shows massive round-off errors, a known

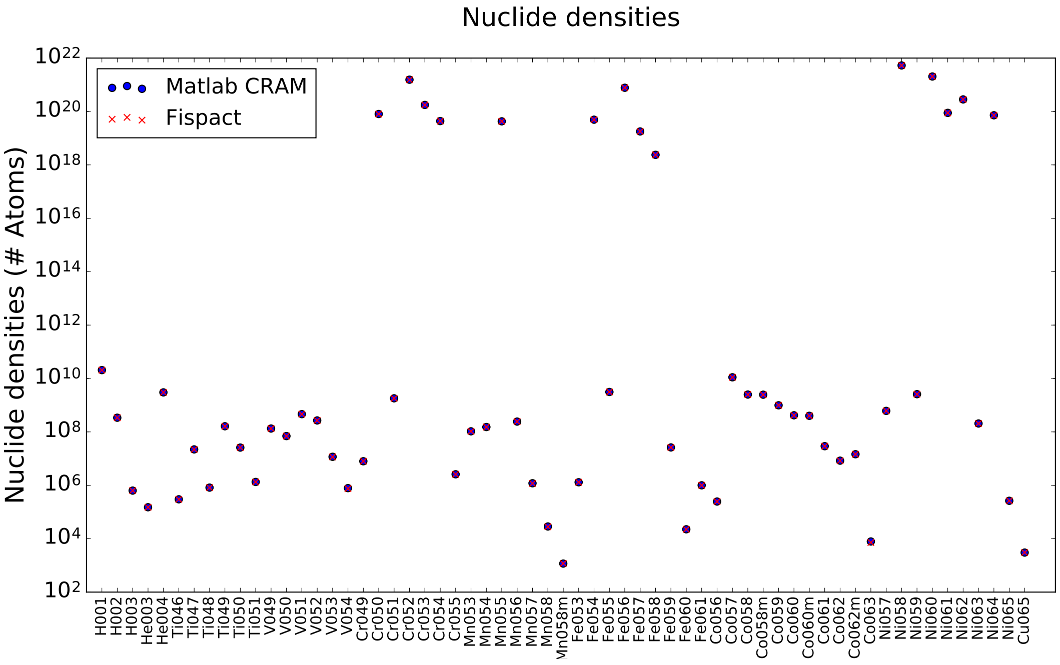


Figure 4.7: FISPACT and MATLAB CRAM inventory calculation show good agreement.

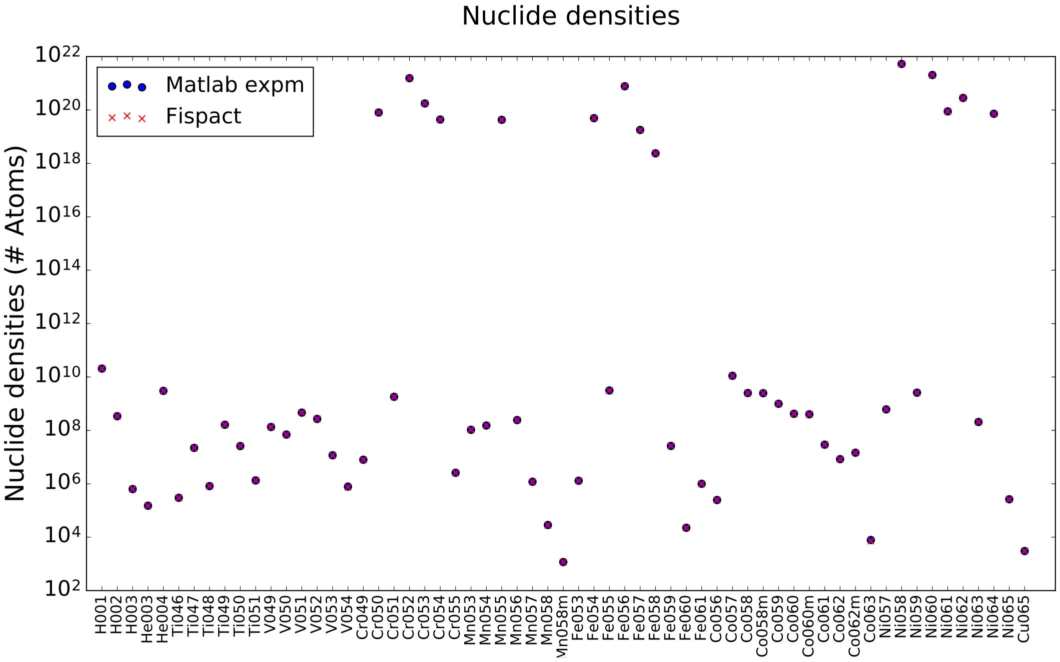


Figure 4.8: FISPACT and MATLAB `expm` inventory calculation show good agreement.

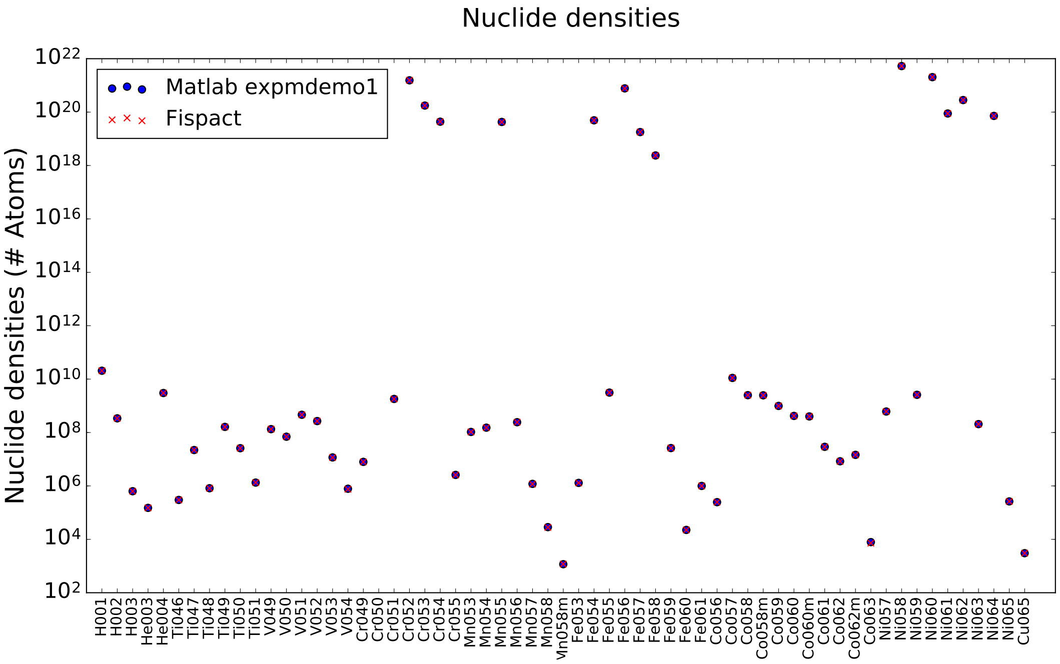


Figure 4.9: FISPACT and MATLAB `expmdemo1` performing scaling and squaring show good agreement.

issue for this method, leading to wildly wrong results. The `expmdemo3` routine, which calculates the matrix exponential by the eigenvalue method, performs poorly when the activation matrix **A** does not have a complete set of linearly independent eigenvectors. The calculated nuclide densities are therefore inaccurate and the only nuclide densities which agree with the FISPACT nuclide densities are for the nuclides that are present in the initial nuclide inventory and therefore the nuclides with a higher density.

These results show the difficulties associated with numerical quality of some of the different methods of calculating the matrix exponential as mentioned earlier. However these results confirm that the activation matrix creation step of the simulation is correct as the final results calculated by some of the methods are in agreement. It should also be mentioned that in all of these examples, inventory densities with less than 1000 atoms are considered statistically insignificant and so not included in the MATLAB plots. This is because FISPACT doesn't output nuclide densities less than this, as set by the MIND keyword in the FISPACT simulation.

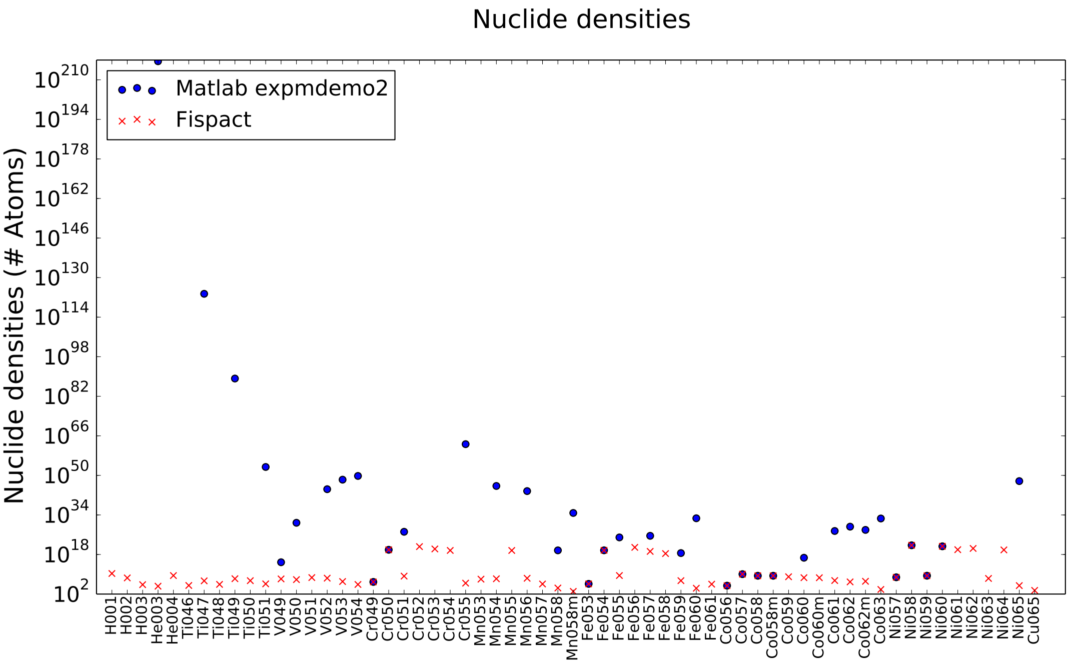


Figure 4.10: FISPACT and MATLAB `expmdemo2` calculating the Taylor series show poor agreement. Large errors occur for almost all nuclides due to round-off errors.

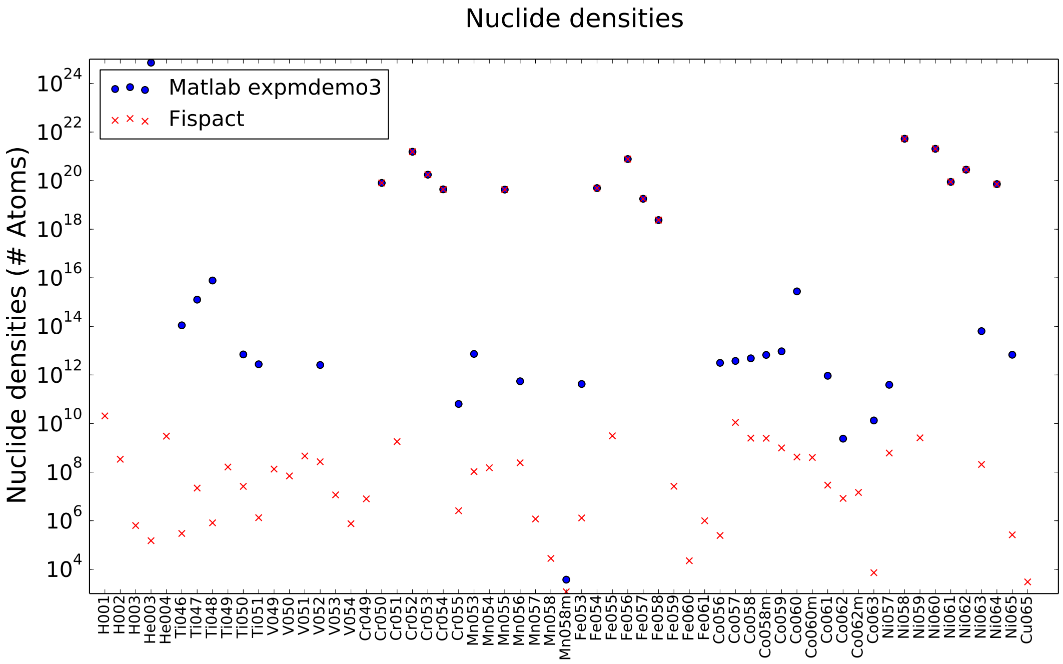


Figure 4.11: FISPACT and MATLAB `expmdemo3` show poor agreement for many nuclides. Nuclides that are present in the initial nuclide inventory agree well where as nuclides present after the irradiation step agree less well.

4.6.2 FISPACT vs. GPU

CUDA kernels have been written to perform the operations contained in the MATLAB code in Fig. 4.6 (see Appendix B.2). The MATLAB operation $\mathbf{A}\backslash\mathbf{b}$ is a shorthand for finding the solution of a system of linear equations $\mathbf{Ax} = \mathbf{b}$. To do this on the GPU a library routine is used. A few options have been explored here to evaluate their performance, specifically the MAGMA library [162] [163] and the CuSparse library [164].

Using MAGMA

The MAGMA routine used here is `magma_zgesv_gpu` [165], which does LU decomposition with partial pivoting and row interchanges to factor $\mathbf{A}$ as $\mathbf{A} = \mathbf{P}*\mathbf{L}*\mathbf{U}$, where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is unit lower triangular, and $\mathbf{U}$ is upper triangular. The factored form of $\mathbf{A}$ is then used to solve the system of equations $\mathbf{Ax} = \mathbf{b}$ as:

$$\begin{aligned}
&Ax = b \Longrightarrow PLUx = b \\
&LUx = P^{-1}b \qquad \text{(as P is orthogonal an inverse exists)} \\
&Ly = P^{-1}b \qquad \text{(first solve for intermediate result y)} \\
&Ux = y \qquad \text{(second solve for x)}
\end{aligned}$$

As can be seen from Fig. 4.12 this routine provides good agreement with FISPACT. However, it does not exploit the sparsity structure of the matrix and there is wasted space on the GPU. Further as the routine is a hybrid GPU-CPU routine [166] it is synchronous and there are many synchronisations and data copies between the GPU and the CPU, hence it is slow for matrices of the size encountered for nuclear burn up calculations, which are size 2604 × 2604 if we include all the isotopes from the ENDF nuclear data library which have non-zero MAT number in the simulation. This routine is optimized for very large dense matrices where the synchronisations and data copies between the CPU and GPU are negligible and the copies can be overlapped with GPU work. Hybrid algorithms are not as efficient for small sized problems as they are for large sized problems. However, if multiple CPU-GPU pairs

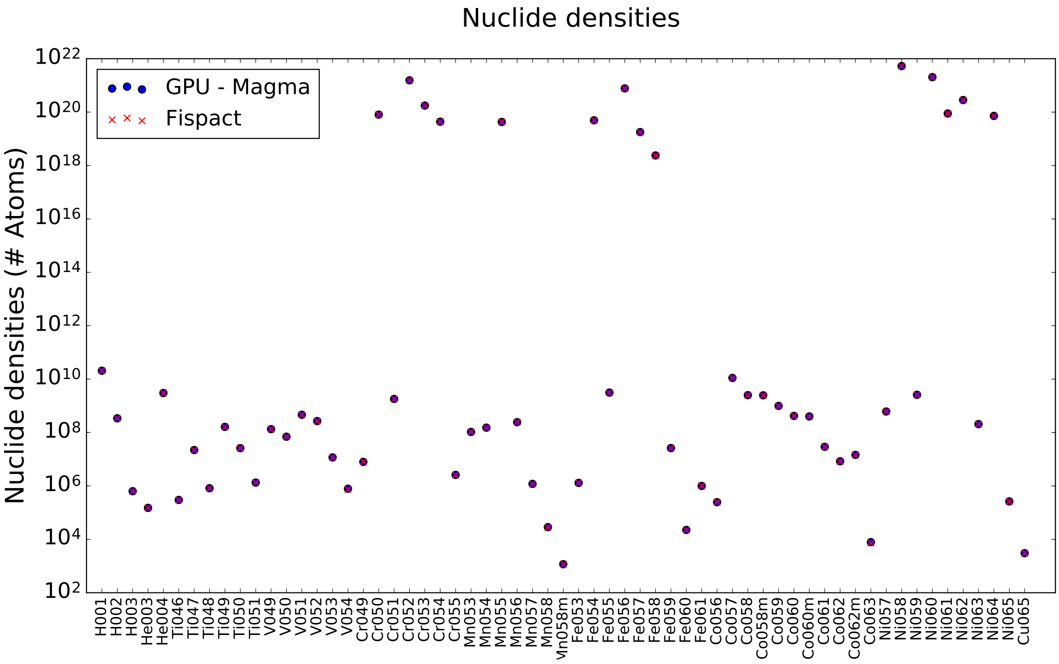


Figure 4.12: FISPACT and GPU inventories using `magma_zgesv_gpu` to solve the system of linear equations show good agreement with the GPU and FISPACT nuclide densities overlapping.

are available then tiled LU factorisation can be performed on the multi-CPU-GPU paired system to give a performance benefit [167].

To improve on this situation for the particular problem encountered in burn up calculations, where we want to solve many independent matrices of size 2604 × 2604, the possibility of using `magma_zgesv_batched` [165] to solve many matrices simultaneously, for example all the time steps for one matrix, was investigated. This routine is executed entirely on the GPU and worked well for smaller-sized test matrices, for example matrix dimensions of size less than or equal to 512 [168]. However, including all the non-zero MAT isotopes in the full model means each matrix has size 2604 × 2604 and the benefit from using `magma_zgesv_batched` is lost. These matrices are too large to be solved efficiently with the batch processing routine and need to be solved serially, but as discussed above this does not give the required performance either. Additionally, the size of the full problem, including all 21 time steps at the same time, is larger than the available 2GB GDDR5 memory on the GTX 680 card and so the whole problem will not fit on the GPU (although

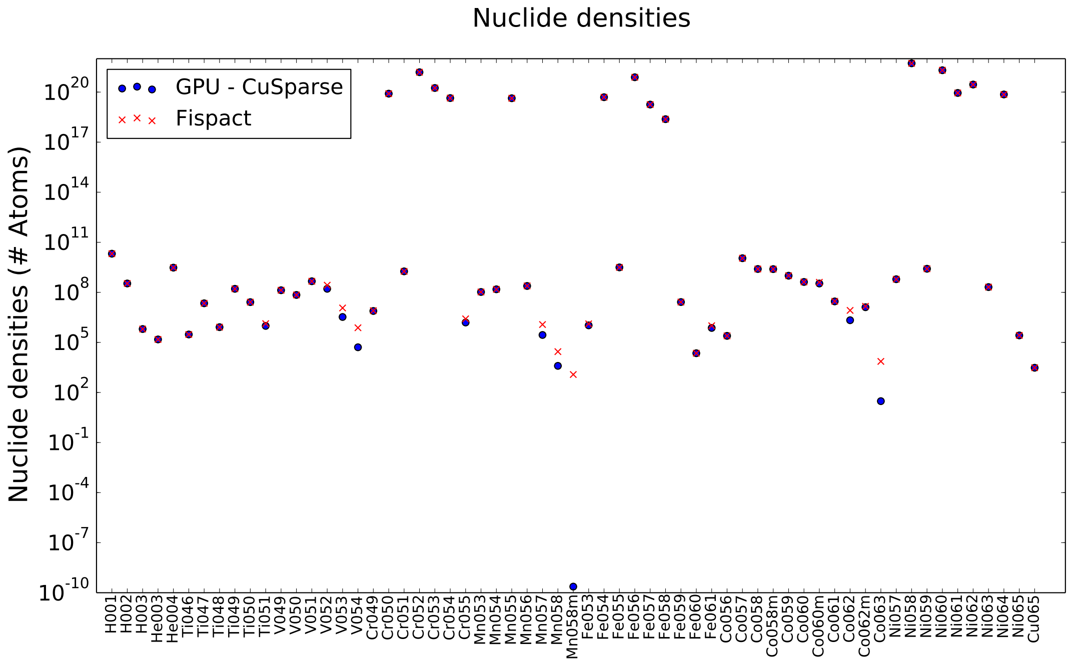


Figure 4.13: FISPACT and GPU inventories using CuSparse to solve the system of linear equations show good agreement for some nuclides but poor agreement for others.

could be easily alleviated by moving to a larger GPU).

Using CuSparse

As an alternative method, to exploit the sparse nature of the activation matrices, the use of the CuSparse library for performing LU factorisation of a sparse linear system was explored. The activation matrix is first converted to compressed sparse row (CSR) format and $\mathbf{A}$ is factored as $\mathbf{A} = \mathbf{L} * \mathbf{U}$ and solved similarly to the MAGMA routines. Unfortunately, this method does not give good agreement for all nuclides and is not in good agreement with FISPACT as can be seen in Fig. 4.13. It is faster than the MAGMA routine however which will be discussed in the next section.

4.7 Acceleration of activation matrix creation and CRAM solve

The acceleration benefits a GPU based system can provide are realised primarily because each cell has an independent activation matrix $\mathbf{A}_b$ to solve. So each GPU in a system can solve a different matrix simultaneously. Additionally, each GPU could potentially solve more than one matrix using CUDA streams, as long as there is enough memory on the GPU to solve the full problem. For example if three activation matrices could fit on one GPU, three CUDA streams could run concurrently to solve them simultaneously. Then if there were four GPUs in a system, 12 matrices could be solved simultaneously. Unfortunately, this cannot be done with the MAGMA routines as they are synchronous routines, meaning there is no acceleration provided by using CUDA streams as each stream has to wait for the previous stream to finish before it can start. In this case, in a four GPU system, only four matrices can be solved simultaneously by launching each GPU with an OpenMP thread. CuSparse routines could potentially benefit from using CUDA streams as they are asynchronous routines, but as this method has been shown to be not numerically sound for burn up activation matrices it was not investigated.

4.7.1 MAGMA acceleration

Fig. 4.14 shows the performance of the GPU CRAM routine using the MAGMA library to solve the system of linear equations, compared to the FISPACT inventory calculation for one irradiation step and 21 cooling steps. The MAGMA CRAM routine is much slower than the FISPACT inventory calculation. As already discussed, the `magma_zgesv_gpu` routine used is a hybrid GPU-CPU routine, meaning it is synchronous and there are many copies to and from the GPU which limits performance. This routine does however provide a performance benefit over CPU-only implementations when the matrices to be solved are large. For the matrices in burn up calculations which have size 2604×2604, the FISPACT inventory calculation itself is very quick, approximately 2 seconds for each cell M. This is if the collapsed cross-sections have already been computed and the inventory calculation is done in

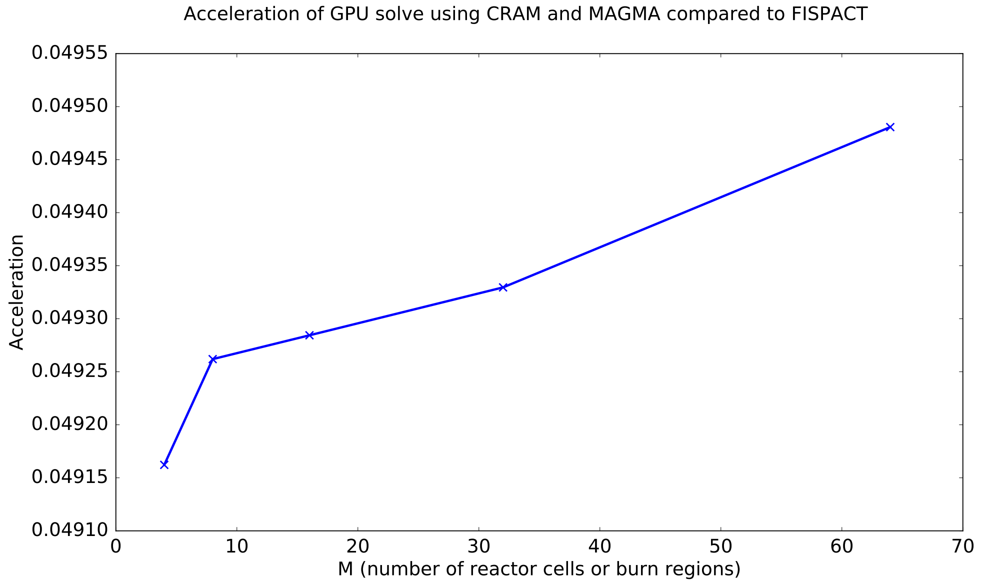


Figure 4.14: The performance of the GPU MAGMA CRAM over the FISPACT inventory calculation. The FISPACT inventory step is actually much faster than the GPU CRAM method using the MAGMA library.

isolation. The runtime of FISPACT is dominated by the first step, the nuclear data cross-section collapse. Fig. 4.15 shows the overall performance for the full GPU code and FISPACT simulation. The GPU code has similar overall performance to FISPACT and is only a disappointing 1.5 times as fast as the FISPACT code. The huge speed-up seen in the GPU cross-section collapse has been largely negated by the slow MAGMA CRAM routine.

4.7.2 CuSparse acceleration

The CuSparse CRAM acceleration over the FISPACT inventory calculation for one irradiation step and 21 cooling steps is shown in Fig. 4.16. The CuSparse method performs better than the MAGMA method and is comparable to FISPACT in terms of performance. However, the GPU performance is still disappointing and these results indicate that it might be better to only perform the cross-section collapse on the GPU as this gives a huge performance increase, and then feed the collapsed cross-sections to FISPACT to calculate the inventory.

The total acceleration provided by the full GPU code is shown in Fig. 4.17. This

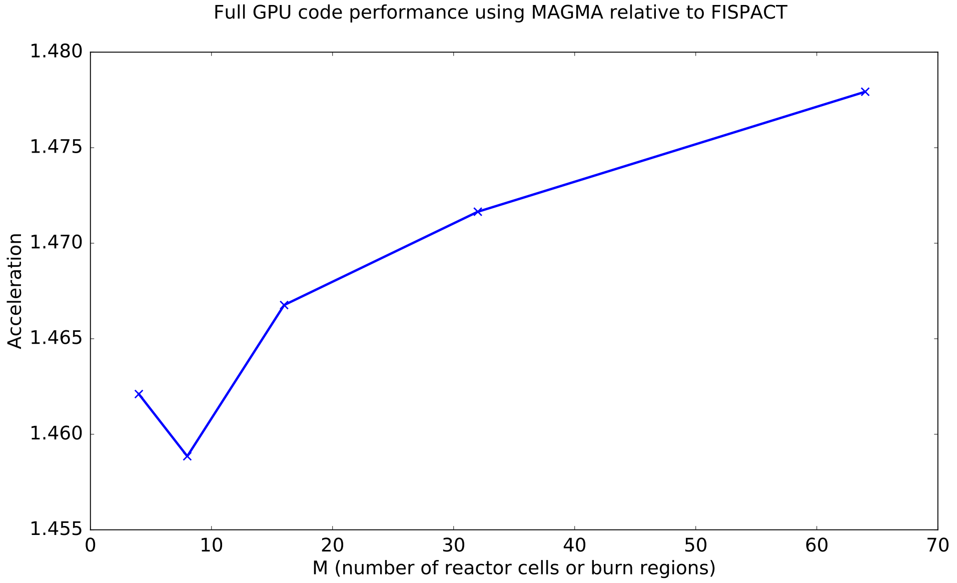


Figure 4.15: The performance of the full GPU MAGMA CRAM over FISPACT. All of the benefit provided by the fast cross-section collapse on the GPU is offset by the poor performance of the MAGMA CRAM method such that both methods perform approximately equally.

appears to level out and provide an acceleration just short of 30x over FISPACT for many burn zones. Again this performance gain is solely due to the accelerated cross-section collapse on the GPU.

4.7.3 Linearity of activation matrix creation and CRAM solve parts of the code

The run time of both the activation matrix creation and CRAM is proportional to the number of burn zones (M) as can be seen from Fig. 4.18 and Fig. 4.19.

The activation matrix creation is performed on the CPU as there is the potential for race conditions on the GPU where different threads representing different reactions would need to access the same data element and modify it. To avoid this, CUDA atomics could be used to ensure threads were acting on the value stored in that memory location but performance would be significantly impacted. For this reason it was considered that the best option is to perform the activation matrix creation on the CPU. As such, each activation matrix for each burn zone is created

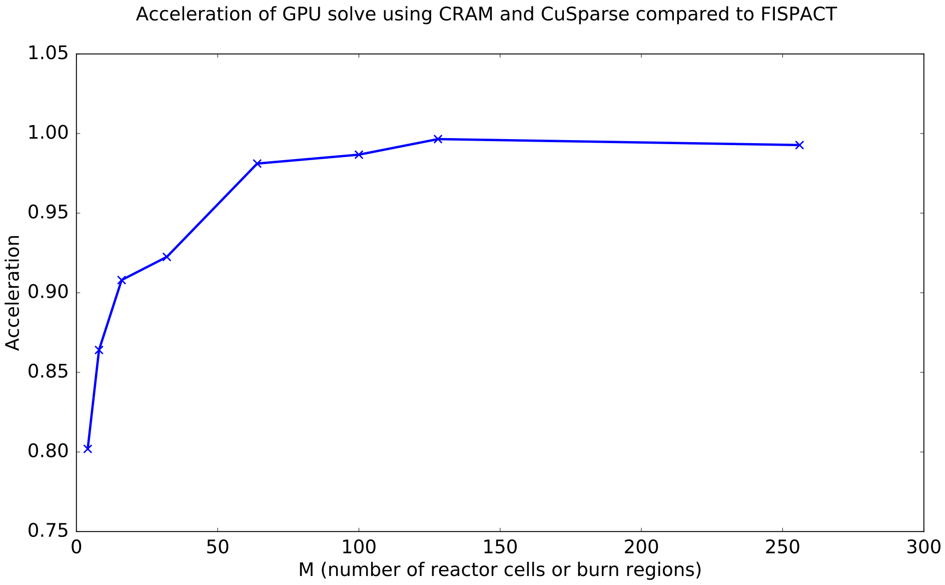


Figure 4.16: The performance of the GPU CuSparse CRAM over the FISPACT inventory calculation. The two methods are comparable in terms of performance.

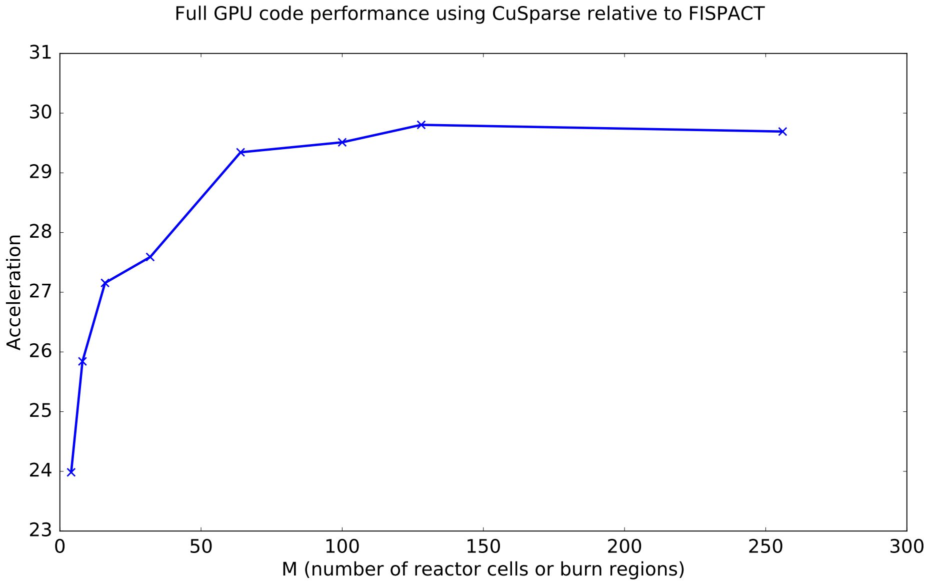


Figure 4.17: The performance of the full GPU CuSparse CRAM over the FISPACT. The GPU code has accelerated the solve by 30x.

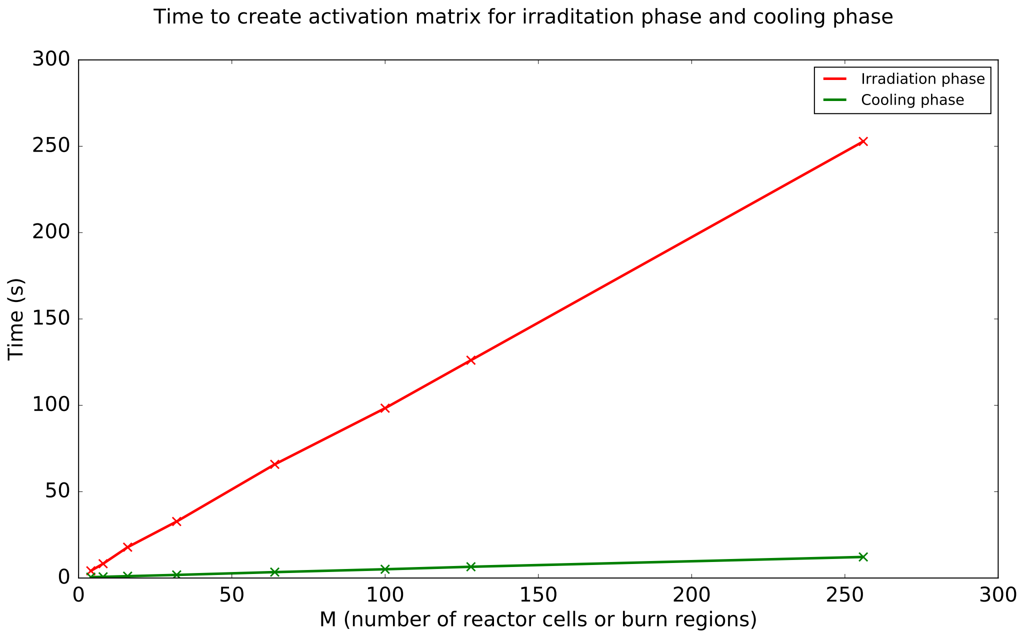


Figure 4.18: The run times for the irradiation and cooling phases of the activation matrix creation. The cooling phase is quicker as only the decay constants are included in the matrix creation.

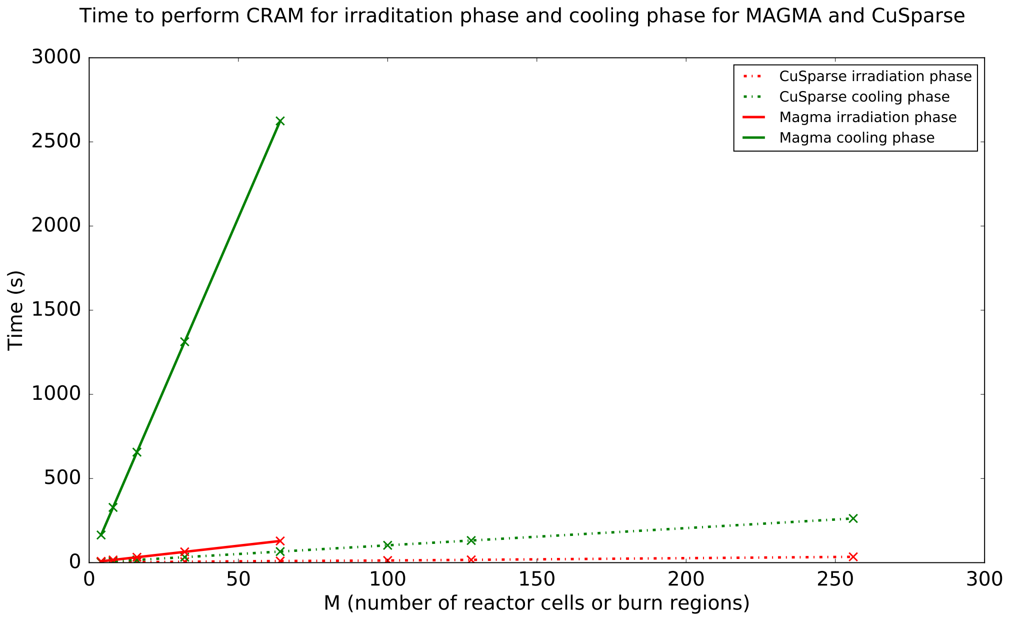


Figure 4.19: The run times for the irradiation and cooling phases of the CRAM solve for both MAGMA and CuSparse methods. The irradiation phase is quicker as this was just one time step in the FNS_INCONEL simulation, whereas the cooling phase has 21 time steps to calculate.

sequentially and therefore we see a linear relationship as M increases. This situation could be improved somewhat using OpenMP to parallelise the creation of several activation matrices for several burn zones simultaneously on the CPU machine.

There is a similar effect occurring for the CRAM solver. Each activation matrix is solved on one GPU at once by launching an OpenMP thread for each GPU. In the cluster used, there are four GeForce GTX 680 cards, so this has been parallelised so that each of the four GPUs is solving one matrix for a different burn zone simultaneously. This is shown in Fig. 4.20 for a simulation considering four burn zones so that M=4. Running the simulation on one GPU so that each matrix is solved sequentially takes 3.9 times as long as running on four GPUs for the MAGMA routine, and 3.2 times as long as running on four GPUs for the CuSparse routine. Theoretically, one would expect the runtime to be four times faster running on four GPUs. In reality there is a slight overhead when switching between GPU contexts which is more noticeable in the CuSparse case as the run times are much shorter. This effect is negligible in the MAGMA case as the run times involved are large. It is clear that on a bigger system with more GPUs, this part of the simulation would be significantly faster and is proportional to the number of GPUs in the system. Massively parallel GPU machines, such as the RAL Emerald cluster [169] with 197 NVIDIA GPUs (two K80, three K20 and 192 M2090 GPUs), would be well-suited to such a calculation.

4.8 Absolute and relative tolerance settings for FISPACT to use in LSODES solver

Looking at the relative error between FISPACT and some of the apparently successful methods, see Figs. 4.21 - 4.23, the agreement between FISPACT and the new method is only 10% for some nuclides. This was found to be because the default tolerance settings which are passed to the LSODES solver to control the convergence of the solution are not strict enough for this FISPACT simulation. The default settings have an absolute tolerance $atol = 1e^{4}$ and relative tolerance $rtol = 1e^{-3}$ and give good accuracy for the major constituents of the initial inventory but may give

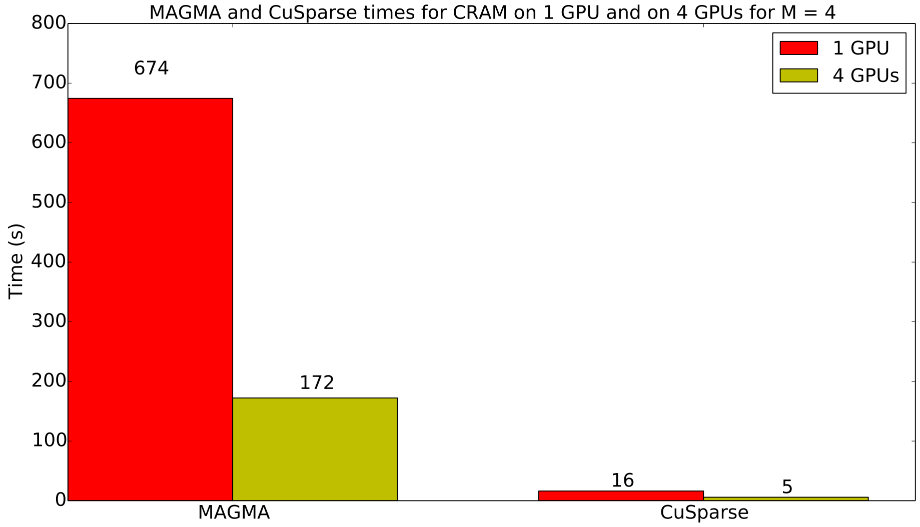


Figure 4.20: Running on a multi-gpu system increases performance as each GPU can solve it's own independent matrix simultaneosuly.

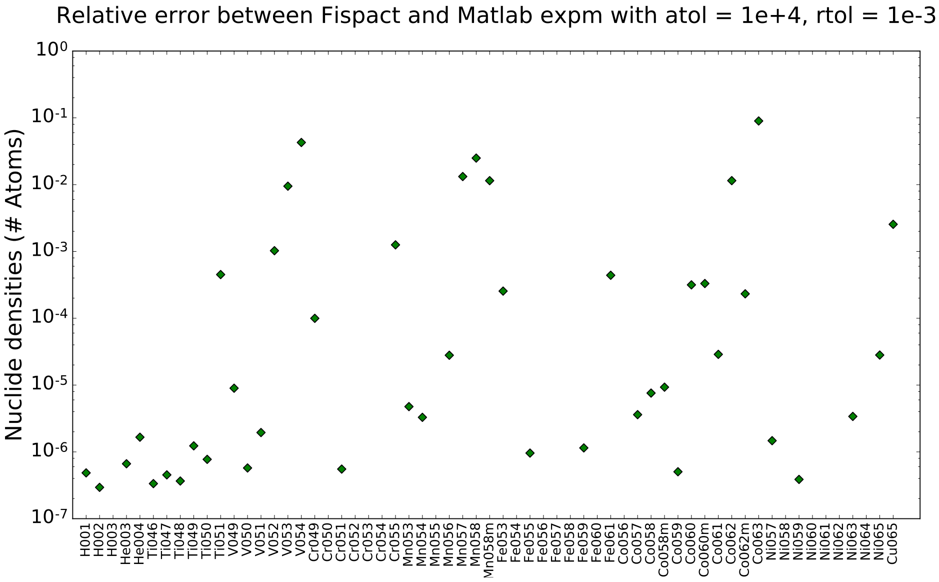


Figure 4.21: Relative error between FISPACT and MATLAB expm routine with default tolerance settings.

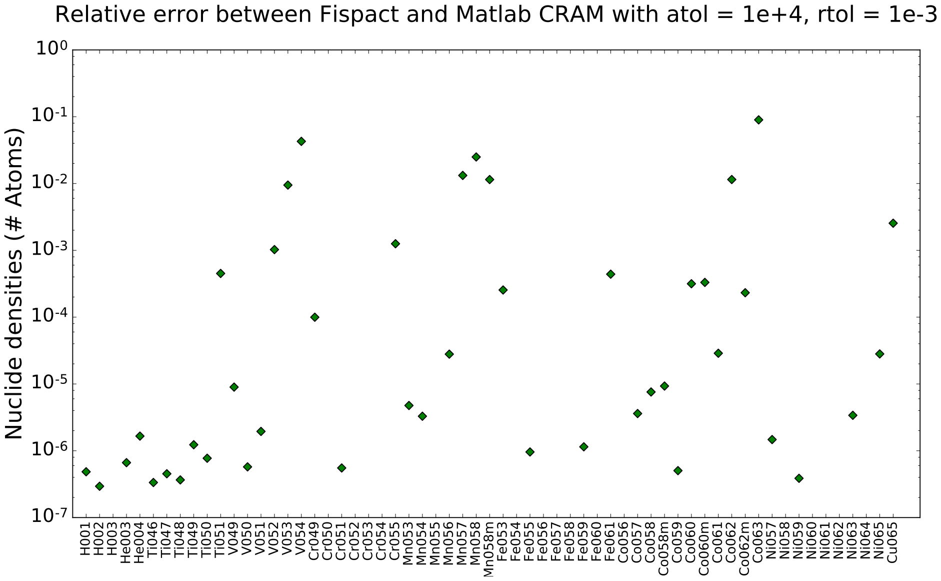


Figure 4.22: Relative error between FISPACT and MATLAB CRAM routine with default tolerance settings.

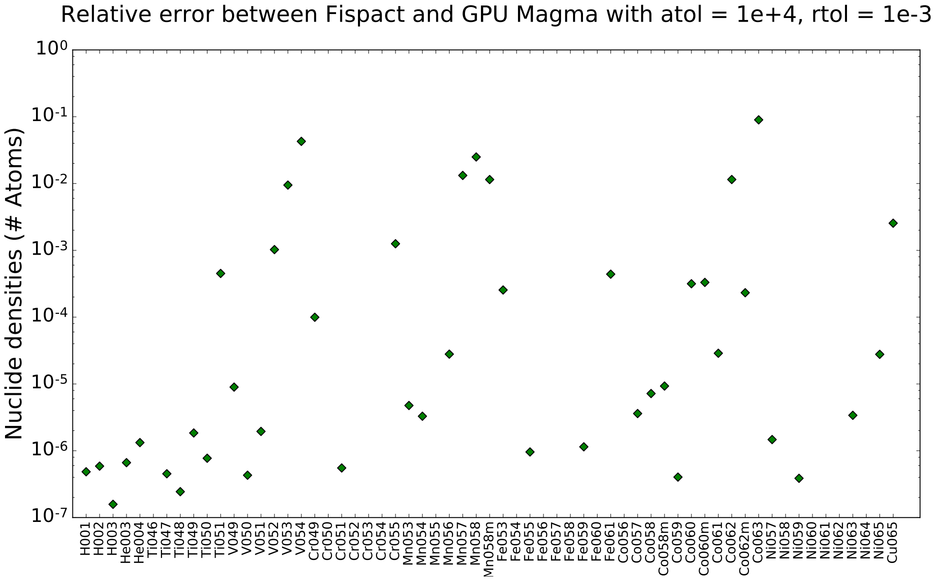


Figure 4.23: Relative error between FISPACT and GPU MAGMA routine with default tolerance settings.

poor results for the minor constituents which may lead to inaccurate predictions of radiological quantities when minor constituents are dominant in radiological activity [133]. Tighter tolerances must be used to ensure that the significant radiological outputs have converged such that the inventory calculation is accurately calculated. Setting stricter tolerances will increase the runtime of this part of the simulation, particularly for complex simulations with many nuclides present. Table 4.2 shows the different tolerance settings tested and the run time for this part of the simulation associated with each tolerance. For each tolerance setting, the nuclide density for ^{63}Co as calculated by FISPACT is also shown, as is the relative error with the nuclide density calculated by the GPU code. For this example the runtime has increased by approximately 40% from the default tolerance setting to the strictest but as the runtime is only a few seconds, this effect is negligible on the overall solution time. The relative error between FISPACT and the GPU CRAM method using MAGMA for $atol = 1e^{-2}$ and $rtol = 1e^{-9}$ is shown in Fig. 4.24 and has reduced to less than 0.01%. After performing a pathways analysis for the FISPACT simulation, it was found that most of the nuclides with relative error greater that 0.001% are short-lived dominant nuclides and this may be a reason for the larger relative errors for these nuclides. All nuclides with relative error less than 0.001% are long-lived. Fig. 4.25 shows both the relative error at the default tolerance setting and at the stricter tolerance setting to illustrate which nuclides have changed and by how much. Additionally, Table 4.2 shows that we get reasonable agreement between FISPACT and the GPU without having to use the strictest tolerance settings.

4.9 Final words on a GPU-based Bateman solver

A GPU code has been designed and tested against the FNS_INCONEL example in FISPACT and also been benchmarked against two other FISPACT examples, the FNG_Nickel (Frascati Neutron Generator from Nickel foil irradiation) and 30keV Maxwellian (an astrophysical example with average stellar abundances). Similar acceleration has been observed and the quality of the results has been maintained for these further two examples. Unfortunately, a feasible GPU solution for the

Table 4.2: Different tolerance settings for LSODES solver showing effect on runtime, nuclide density and relative error for ^{63}Co

atol	**rtol**	**Time (s)**	**Nuclide density ^{63}Co**	**Relative Error ^{63}Co**
1e+04	1e-03	1.97	7.21988e+03	8.97579e-02
1e+03	1e-04	2.08	7.67467e+03	2.51803e-02
1e+02	1e-05	2.13	7.86620e+03	2.18784e-04
1e+01	1e-06	2.21	7.87074e+03	3.57859e-04
1e+00	1e-07	2.26	7.86898e+03	1.34579e-04
1e-01	1e-08	2.48	7.86866e+03	9.39169e-05
1e-02	1e-09	2.82	7.86873e+03	1.02812e-04

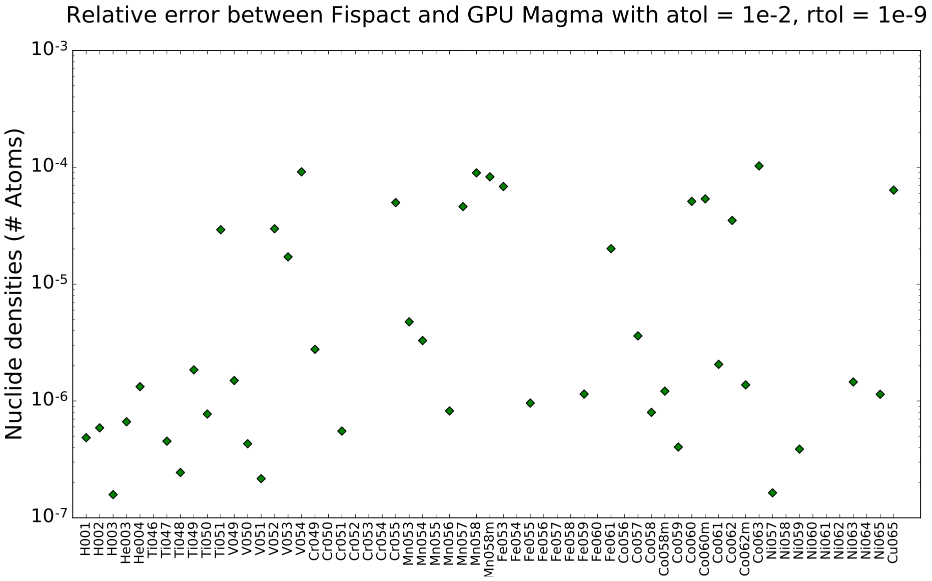


Figure 4.24: Relative error between FISPACT and GPU MAGMA routine with stricter tolerance settings.

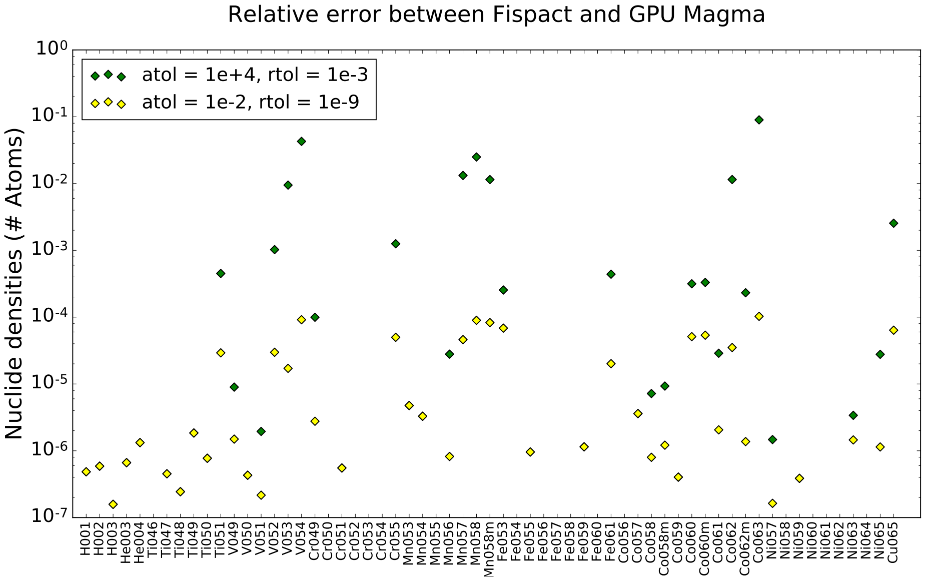


Figure 4.25: Relative error between FISPACT and GPU MAGMA routine for the default and stricter tolerance settings illustrating the improvement in agreement between results a stricter tolerance gives.

CRAM method that is both numerically accurate and fast could not be achieved within the time frame of the PhD. Given more time, further GPU routines to solve sparse systems of linear equations of the type found in burn up calculations could be investigated, for example, the MAGMA routines available through the Sparse Iter package [170].

The GPU code developed is an incomplete model of FISPACT-II and does not include all of the functionality of FISPACT. The activation matrix creation is incomplete and there is data missing from the activation matrix. The self-fission decay lambdas and branching ratios from the nuclear data have been ignored currently in the GPU implementation so these effects are missing from the activation matrix which is then passed to CRAM. Whilst this omission doesn't effect the FNS_INCONEL simulation as there are no actinides present in the simulation, this functionality could be included at a later date to provide the next component of the model and to be able to perform more interesting and fusion relevant simulations.

What has been demonstrated here is that the CRAM method is suitable in general for use in inventory calculations to solve the Bateman equations. In the future it may well be possible to implement CRAM successfully, with a significant acceleration on the GPU, and the work in this chapter provides the ground work for doing so. Additionally, the usefulness of the cross-section collapse on the GPU in step one of the simulation should be mentioned. The ability to do the collapse for many burn zones simultaneously and at an impressive speed is beneficial for many neutronics problems, as is the ability to perform the collapse with several perturbed fluxes at once to account for uncertainties in the nuclear data.

Chapter 5

Matrix multiplication on FPGA-based dataflow engines

Massive, dense linear algebra such as matrix-matrix multiplication is extremely computationally demanding but is widely applied in many scientific computing applications [171] [172]. For example, to clarify and predict the properties of a variety of materials, density functional theory relying on matrix multiplication is used [173]. Massive matrix multiplications are also relevant in matrix equations describing many fusion applications such as studying low frequency, long-wavelength plasmas [174].

New hardware is now available to offload computationally expensive parts of an application to provide an acceleration: the dataflow engine, an FPGA-based reconfigurable architecture providing the flexibility to adjust the architecture to a specific problem. Previous successful work using Maxeler dataflow engines was introduced in Chapter 2 and includes molecular mechanics simulations and the calculation of induced dipoles [89], sorting algorithms such as the bitonic mergesort algorithm [90], stencil computation known to be computationally demanding and prevalent in many scientific areas [91] and a specific finite difference library in the form of MaxGenFD [92]. These examples provide significant improvements in performance relative to CPU-based systems and some have demonstrated positive comparisons with GPU implementations. Additionally, the performance per watt is greatly improved in the dataflow based solutions [175] making them the energy efficient alternative to more traditional approaches.

To explore the capabilities of the dataflow engine, a massive matrix-matrix multiplication problem was chosen due to the high importance it has in many scientific applications, including in plasma physics. Indeed, we have come across matrix multiplications in the SAMI data processing chain in Chapter 3 and the cross-section collapse in the first step of solving the Bateman equations in Chapter 4. In this chapter the performance and energy efficiency of a massive matrix-matrix multiplication on a dataflow engine are compared with other known hardware solutions such as the GPU and many-core architectures running the well developed high performance libraries CUBLAS [148] and Intel MKL [176].

5.1 Matrix multiplication on FPGAs

Recently there has been an increased interest in matrix-matrix multiplication on FPGAs. Most of this work has focussed on multiplying small matrices which is a limitation due to the available resources on the FPGA, i.e. the matrices have to fit in the small on-chip memory. One study [177] considers multiplying two four-by-four matrices using the Simulink Xilinx platform [178]. A parallel implementation uses a lot of hardware resources and as such means it is difficult to perform larger matrix multiplies. An alternative design follows a serial approach and uses less resources but takes longer to perform the calculation. Other studies [179] [180] perform multiplications reaching 200*200 and 1024*1024 respectively, as the maximum size considered.

This is a well studied field, with many people working on different optimisations to bring about the best performance. However much of the previous work has required a deep understanding of the FPGA hardware and hardware description languages like VHDL or Verilog, such as in [181] which considers a blocking algorithm that allows data re-use on-chip whilst the full matrix is stored in off-chip memory. Another study [182] chooses to focus on optimisations which improve the energy performance in addition to latency and throughput, but again the approach requires a detailed knowledge of the underlying FPGA hardware.

The aim with the relatively new Maxeler dataflow approach is to abstract the fundamentals from the programmer and provide a programming environment that is quick to adapt to and more familiar to most research scientists. It will therefore be accessible to a larger user base within the scientific community and quickly gain traction in the field of high performance computing on FPGAs.

5.2 Custom dataflow engine (DFE) implementation

To explore the capabilities of the DFE a massive matrix-matrix multiplication problem was chosen due to the high importance it has in many scientific applications. Considering a matrix-multiplication $C = A * B$, where A = [N, K], B = [K, M] and C = [N, M], the algorithm used for the dataflow implementation, in terms of how the data is presented to the dataflow engine, follows the optimised C approach:

```
for(int n=0; n<N; n++){
  for(int k=0; k<K; k++){
    for(int m=0; m<M; m++){
      C[n][m] += A[n][k]*B[k][m];
    }
  }
}
```

This is optimised as the orders of the loop k and m have been swapped when compared to a naive implementation. The order of the elements of A and B that are multiplied together are contiguous so that the order that the elements of C are computed are also contiguous. This means there are fewer cache misses as when the first read request to B is made the cache is filled with the row-wise elements of B. Then when the next read request to B is made, there is a cache hit as the element is already in cache and the multiplication can be done immediately, without having to go to main memory for the data. If the orders of loop k and m hadn't been swapped so the next element required from B is from the next row M elements away we have a cache miss.

For the DFE kernel code, the dimensions of the matrices, N and K are passed as scalar inputs from the CPU so there is the ability to change the dimensions of the matrices without having to recompile and build the design. The compilation and build process for FPGA based systems is lengthy and can take up to 24 hours for large designs because of the optimisations and timing constraints that must be met when producing the bit stream for the FPGA. Fortunately, there is a simulation environment for the design and testing phase which is much faster where the FPGA code is simulated in the CPU. The dimension M is fixed and is related to the pipeline depth of the kernel. The kernel pipeline is the chain of operations that performs each multiplication and addition for each data element, the result of which is then passed back into the start of the kernel pipeline to be used for the next addition. In such a way the multiplication and accumulation performs a matrix multiplication. The pipeline depth is greater than one unlike for the simple kernel shown in Chapter 2 because floating point arithmetic cannot be performed in one clock tick, the fundamental unit of time governed by the FPGA clock speed. The minimum required value of M for the custom design to allow a successful build was found to be M=1440. This is used in the kernel to set the stream offset and specifies the number of data elements to offset by to link the current result element to the correct accumulation column. For M $\leq$ 1440, the design fails as the cyclic loop in the kernel has a latency greater than zero. In fact there is a latency of 15 in this pipeline, 12 for the floating-point adder, 1 for the ternary-if operator and 2 for the multiplication so the result of each multiply-add is available 15 ticks after it starts. In each kernel clock tick we are operating on data the size of the burst width i.e. 96 elements of data, so we need to have a stream offset of $15 \times 96 = 1440$ to successfully schedule the dataflow and to provide the correct final result.

The input array (or stream) A is streamed over PCIe and a new input is read only at the start of each new row of B. The stream B is streamed from LMem which is a large off-chip memory as discussed in Chapter 2 and illustrated by Fig. 2.6. The LMem operates in burst sizes of 3072 bits meaning 96 32-bit floats can be read and 96 multiplications can be done at once. The body of the kernel, performing the matrix-matrix multiply consists of a hardware loop. At the head of the loop, if the first row of C is being computed so that we are starting the accumulation,

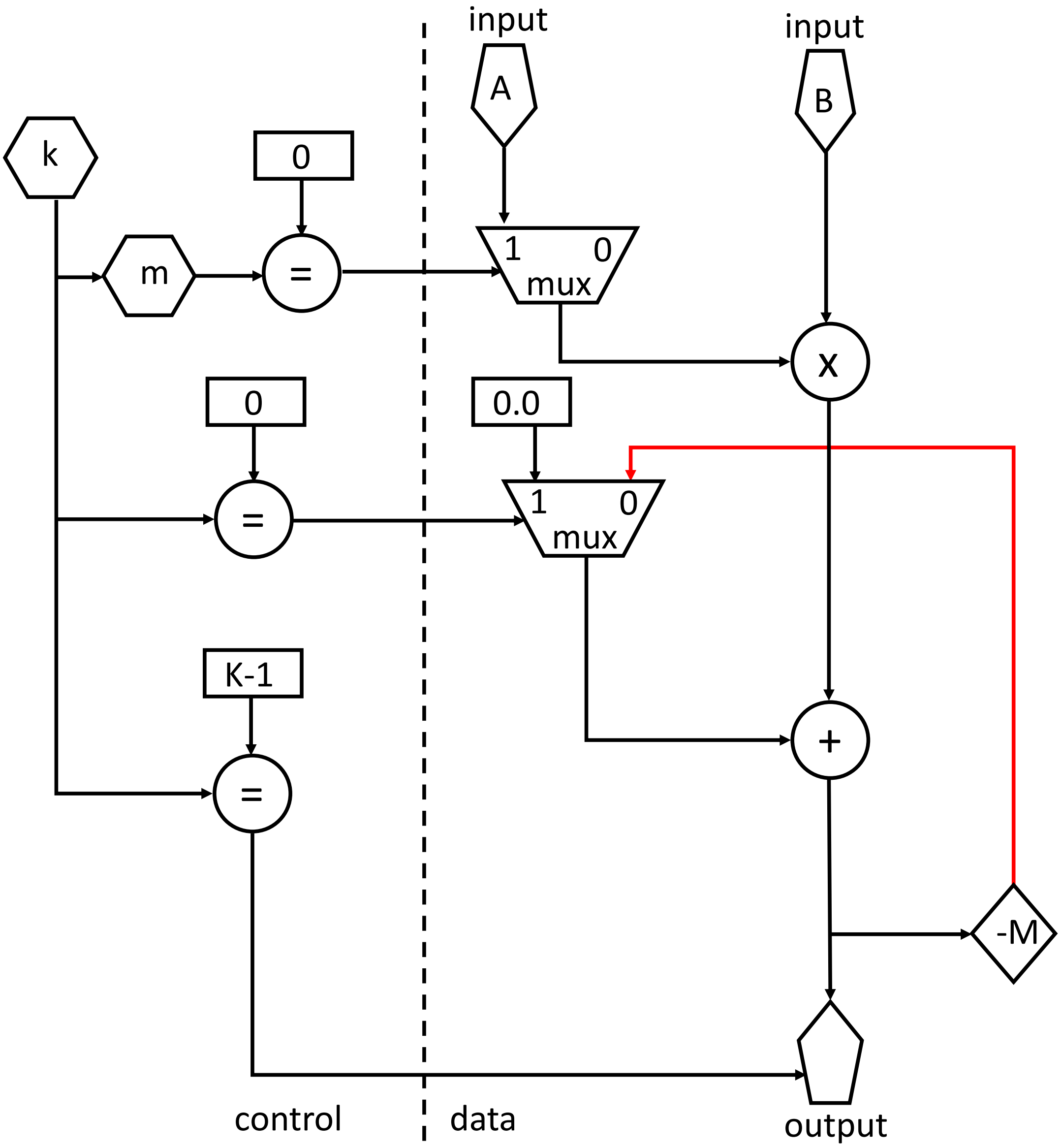


Figure 5.1: Simplified kernel for the custom matrix multiplication design showing the controlled input, hardware loop and control counters. An element from A is read only when the first element from a row of B is read, i.e. when counter m=0. An element from B is read on each clock tick. The current A and B elements are multiplied and for the first m ticks (whilst counter k=0) 0.0 is added, otherwise the value being carried around the loop is added. This result is then offset by M elements (i.e. the length of a row of B) to be accumulated with the next multiplication for that column. Only when the last row of B is being read as input (i.e when counter k=K-1) can the final result start being outputted. This describes the computation for the first row of the matrix C.

the value 0.0 is chosen. Otherwise the accumulated sum being carried around the loop is choosen. The multiplication occurs and the sum is added to produce a new value and the foot of the loop connects this new value to the head of the loop using a stream offset. Finally the correct accumulated result is output to LMem when the end of a row of stream A is reached. A simplified schematic of this kernel is shown in Fig. 5.1 showing the counters k and m which count to K (the length of a column of stream B) and M (length of a row of stream B) respectively which control the dataflow, the controlled input A and the hardware loop performing the matrix multiplication.

The kernel describes the computation but a manager is needed to describe the data movement to, from and within the DFE. In the manager the matrix dimensions are initialized and made available for use in the kernel. Input B and the output C are connected to the LMem with a linear 1D access pattern and input A is connected to the CPU. Interfaces for writing and reading the LMem are constructed and the number of clock ticks for the kernel to run for is also set in the manager. Care is needed to ensure that when the data is transferred from the CPU the input and output matrices are buffered so they are multiples of 3072 bits, the burst size.

Compiling the DFE code creates a SLiC function as described in Chapter 2 which is then available to call from the host CPU code. First matrix B is loaded into the LMem before calling the SLiC function which runs on the dataflow engine which streams A from the CPU and reads B from the LMem, performs the matrix-multiplication and writes the result C to LMem. The result matrix C is then read from LMem into CPU memory after the SLiC function returns. A schematic illustrating this data movement is shown in Fig. 5.2 with A streamed directly into the kernel (shown in Fig. 5.1) from the CPU and B and C going via LMem. The host code, kernel code and manager code for the custom DFE matrix multiplication implementation is shown in Appendix C.

Having fixed M and using the kernel and manager as described above, the size of the problem is limited to dimensions satisfying $N * K \leq 372824$ and M=1440. If N*K exceeded this value, the design fails to wait for stream interrupts and the execution fails as a result. Using these upper bounds, multiple different (N, K)

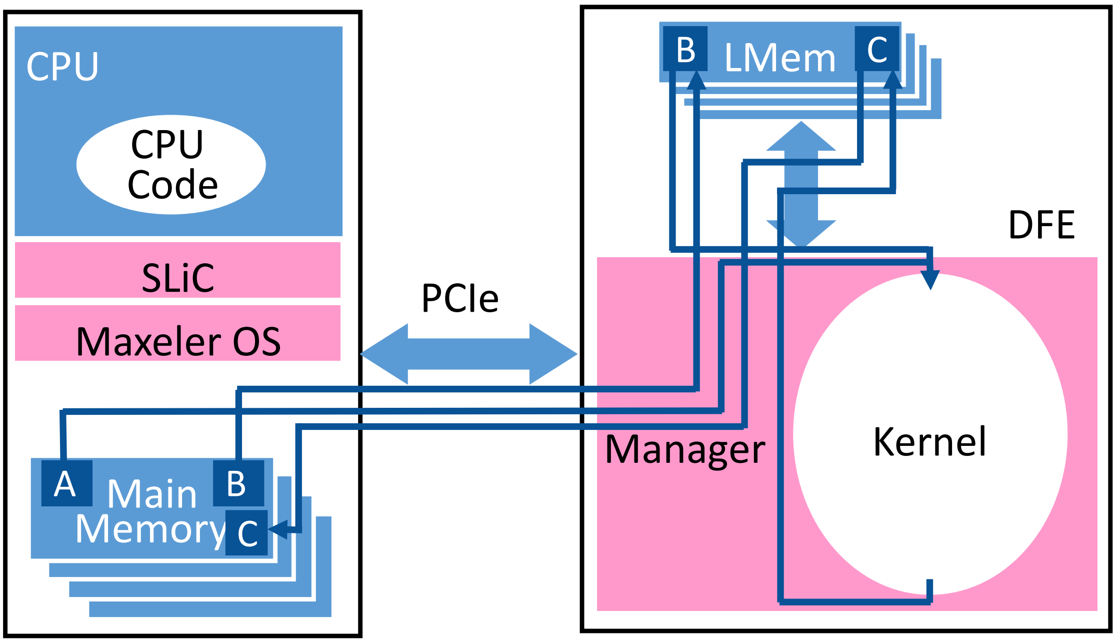


Figure 5.2: Schematic showing the data movement of the matrices A, B and result C between the CPU, DFE LMem and kernel.

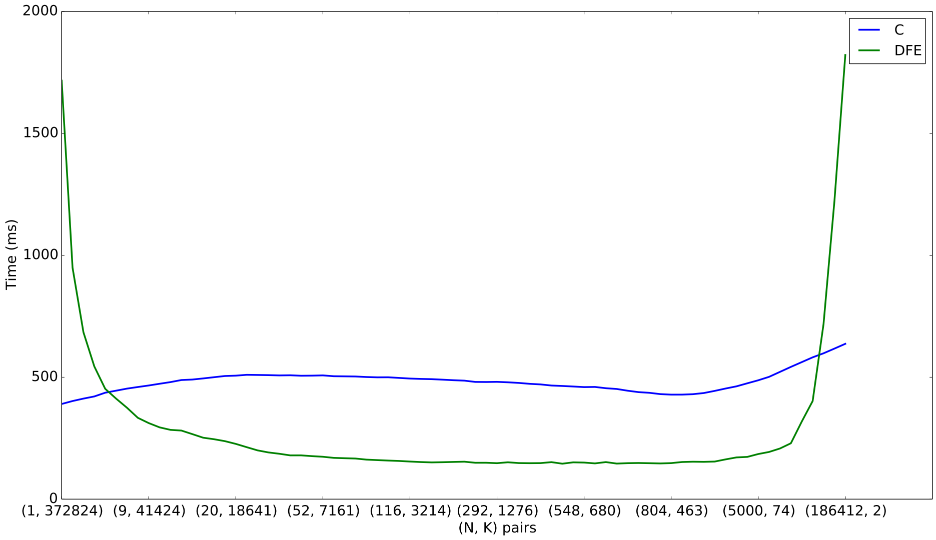


Figure 5.3: The compute times for a CPU and dataflow engine matrix-matrix multiply for different (N, K) pairs specifying the dimensions of the matrix A. The number of columns in the second matrix B is M = 1440. The DFE provides a 3x acceleration over the CPU application, apart from the extreme size configurations where the matrix B specified by the K dimension is short or fat.

combinations for M = 1440 starting from (1, 372824) and ending with (186412, 2) are investigated, fully exploring the parameter space. Fig. 5.3 shows the time taken for both a C implementation and a dataflow implementation in this regime and demonstrates the area of parameter space where it is beneficial to use a dataflow engine over a CPU, i.e. when the dataflow execution times are less than the CPU execution times.

There is a clear region where the dataflow implementation is faster than the CPU implementation and so any of these (N, K) combinations are a suitable choice when looking to accelerate the application. For the following work, three (N, K) pairs; (40, 9320), (300, 1242) and (1000, 372) are considered which sample the parameter space. It is possible to achieve larger matrix multiplications by running the kernel many times to perform a series of smaller matrix multiplications. For example, to perform a (10240, 9320) x (9320, 11520) matrix multiplication, the matrix A is divided into 256 chunks of size (40, 9320) and the matrix B is divided into 8 chunks of size (9320, 1440) resulting in the matrix C being composed of 2048 chunks of size (40, 1440). On one DFE this is implemented as illustrated in Fig. 5.4 (a) for a smaller model with A and B divided into four chunks. The first BChunk is loaded into the DFE LMem and the first AChunk is looped over to give the first CChunk. Then the second BChunk is loaded into the LMem and the process repeated to give the next CChunk and so on until the first N rows of C have been computed. This method scales up to eight DFEs easily. The multiplication can be performed in the same fashion as for one DFE, loading each DFEs LMem with the different BChunks and streaming the same AChunk to the different DFEs so each is computing simultaneously with an independent BChunk. Alternatively the multiplication can be calculated as illustrated in Fig. 5.4 (b). In this method, the same chunk of B is sent to the eight DFEs and different AChunks are streamed to each DFE simultaneously so that the first M columns of the matrix C are obtained. The preferable method is based on performance and is discussed in section 5.4.

There is a fair amount of flexibility in terms of the size of the matrix multiplication as N and K can be varied without having to rebuild the design which can be a very time consuming process. N and K are varied on the CPU and passed

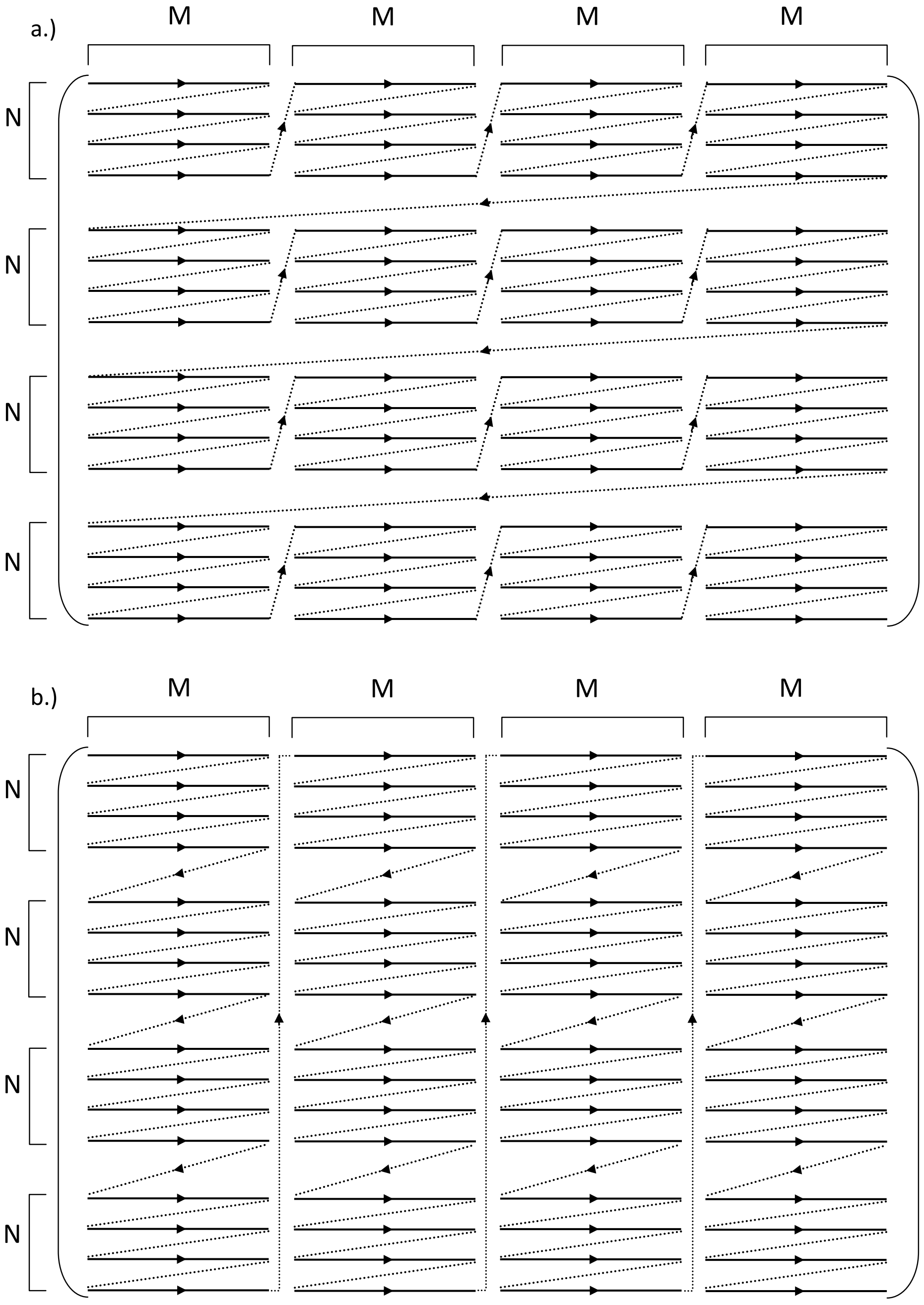


Figure 5.4: A schematic illustrating the order in which the elements of the resulting C matrix are computed for the two different DFE methods, (a) C computed row-wise, and (b) C computed column-wise. Solid lines represent contiguous data elements and dashed lines represent connections to the next contiguous data element.

as scalar inputs to the DFE, M however is hard-coded into the kernel code as the stream offset and can not be varied without the expensive rebuild. The three (N, K) combinations used here are examples only for different orders of magnitude for K and it is clear these can be any value satisfying $N * K \leq 372824$. Any sized matrix multiplication (FullN, K) x (K, FullM) can then be done as a series of (N, K) x (K, M) matrix multiplications where AChunks = FullN/N and BChunks = FullM/M within system memory limitations.

5.3 DFE matrix-multiply using the Maxeler App

Maxeler have developed the Maxeler App Gallery [183] which contains applications developed mostly by Maxeler engineers for tasks such as 1D FFT, 2D FFT, n-body simulation, Jacobi solver and linear regression. There is also a dense matrix-matrix multiplication app available to users of Maxeler hardware. Cloning and building this project results in an highly optimised, matrix-multiplication application.

The Maxeler design works by splitting the matrices up into tiles. If matrix dimensions are not a multiple of the tile size then the matrix is padded to make the dimensions a multiple of the tile size. Splitting each matrix into tiles of size NxN and taking a pair of tiles, one from A and one from B, then N^3 multiplications are carried out on the pair of tiles to give N^2 partial results. Decomposing the matrices into tiles allows for data re-use. Double buffering the tiles allows N multiplications for one read from each matrix into the buffer per cycle. An entire tile from B is loaded into the FMem and a single row of a tile of A is used to compute a row of tile C on every cycle. The remaining rows of A are loaded and the dot-products calculated to give the full tile of C. The matrices need to be rearranged into tile format before computation on the DFE. The dense matrix multiplication app is used below to compare against the custom DFE implementation from section 5.2 and other high performance solutions such as Intel MKL or CUBLAS.

5.4 Results and comparison with other hardware

The DFE dense matrix-matrix multiplication implementation was tested at three different base size (N, K) combinations (40, 9320), (300, 1242) and (1000, 372) with M = 1440. The starting point was to have AChunks = BChunks = 8 and increase both the number of AChunks and BChunks in multiples of eight. As a baseline a serial C implementation was compared to: *(i)* the custom 1-DFE implementation, *(ii)* both methods of implementation for an 8-DFE system, and *(iii)* the implementation provided by the Maxeler App. Also included is 1-thread MKL, 8-thread MKL, and 16-thread MKL implementations (two threads for each of the eight cores), running on an Intel Xeon E5-2670 @ 2.60GHz, and a GPU implementation using a single Tesla K40C running CUBLAS. For the DFE implementations, 1-DFE and 8-DFE solutions were tested on one MPC-X2000 node of the Maxeler machine at STFC Daresbury Laboratory [184] [185] containing eight Maia DFEs each with 48GB LMem [186]. Where relevant, all results include memory copy times to transfer data to and from the accelerator devices, i.e. the DFE and the GPU.

Fig. 5.5 shows the performance of the various implementations for different large sized problems using N = 40, K = 9320 and M = 1440 as the base size (top figure), N = 300, K = 1242 and M = 1440 (middle figure) and for N = 1000, K = 372, M = 1440 (bottom figure). In the top figure, the number of AChunks and BChunks are varied from (8,8) to (128,40) so the problem size varies from [320, 9320] x [9320, 11520] to [5120, 9320] x [9320, 57600]. The custom 1-DFE and 1-thread MKL have similar performance gains. Of the two 8-DFE implementations the method shown in Fig. 5.4 (a) performs better. The MaxApp matrix multiplication implementation performs similarly to the custom 8-DFE implementations and performs slightly better for larger problem sizes. 8-thread and 16-thread MKL perform well and CUBLAS has increasing performance as the problem size increases, up to nearly 600x for the largest matrix multiplication. Similar results have been obtained for the two remaining base cases for N = 300, K = 1242 and M = 1440 (middle figure) and for N = 1000, K = 372, M = 1440 (bottom figure). In these two figures, there are values missing from the custom DFE implementation due to memory limitations on the CPU for the intermediate staging arrays.

The Maxeler App DFE acceleration in the top case is 40x and for the remaining 2 cases is 30x over the serial C implementation. The Maxeler App DFE acceleration compared to 1-thread MKL is between 4-5x whereas 8-threaded MKL outperforms the Maxeler App DFE implementation by 1.5-2x and 16-thread MKL does even better.

These results show a relatively poor compute performance for matrix-matrix multiplication on the DFE when compared to other implementations. The reason for the drastic differences in performance between the DFE and GPU implementation is that the GPU doesn't have to partition the data and do the matrix multiplications as a series of smaller multiplications as is done for the custom DFE, or rearrange the matrices into tiles as in done in the Maxeler App implementation. The Tesla K40C card has 12GB GDDR5 memory so as long as both input matrices A and B and the resulting matrix C fit in memory this is the best solution and performance increases as problem size increases up to this limit. Once the 12GB limit is reached, the problem does need to be split up on the GPU and CUDA streams and concurrency used to process the data. Performance will be reduced slightly as there is time spent arranging the data into the correct chunk format to stream to the GPU.

The DFE implementation is memory bound. For the Maxeler App implementation multiplying square matrices, there are N^3 multiply-accumulates. For the Maxeler App design running at 150 MHz with 1536 DSPs being utilized, with each DSP able to do one multiply-accumulate in one clock cycle, the time taken for a compute bound application is $4.34N^3$ picoseconds. This curve is shown in Fig. 5.6 for square matrices N=K=M, along with the experimental kernel runtimes. As the experimental runtimes are significantly slower than the compute bound runtime the Maxeler App matrix multiplication must be memory bound as the computation is waiting for data from memory. One would expect the experimental runtimes to be slightly slower than the calculated compute times in practice, but the significantly slower experimental runtimes must be down to the slow FPGA memory bandwidth, which is 38.4 GB/s. Matrix multiplication on the GPU is also memory bandwidth bound [187]. This implies the GPU will outperform the DFE simply because it has a faster memory bandwidth, a Tesla K40C has a memory bandwidth of 288

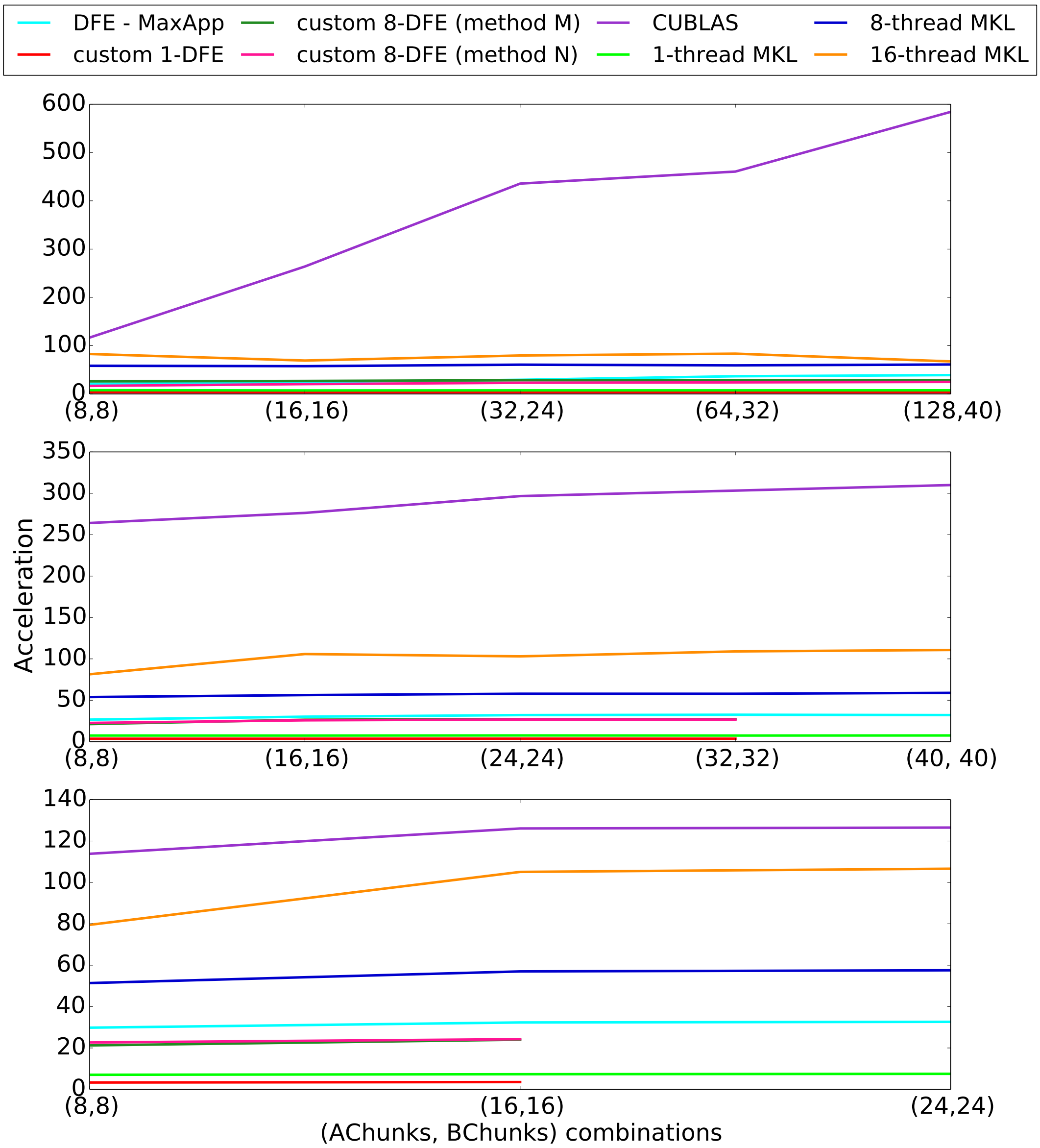


Figure 5.5: The acceleration over a serial C implementation of matrix-matrix multiplication, for increasing matrix sizes. The (AChunks, BChunks) combinations refer to the number of chunks the matrix A and matrix B are split into where the size of a single chunk of A is [40, 9320] (top), [300, 1242] (middle), [1000, 372] (bottom) and the size of a single chunk of B is [9320, 1440] (top), [1242, 1440] (middle) and [372, 1440] (bottom).

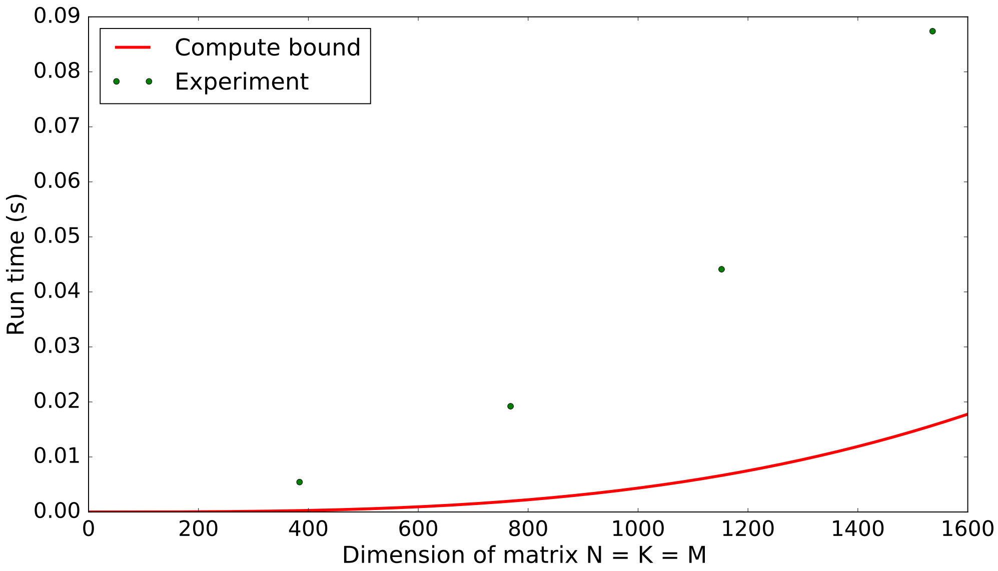


Figure 5.6: The curves for the experimental runtime and compute bound $(4.34N^3)$ runtime for square matrices, N=K=M. Matrix multiplication on the DFE is memory bound as the experimental runtime is slower than the compute time.

GB/s which is 7.5 times faster than the DFE memory bandwidth. Whilst the faster memory bandwidth doesn't account for all of the better performance, the GPU CUBLAS implementation is also a highly optimised library that has been developed and improved over many years, as has the Intel MKL library. The Maxeler App dense matrix multiplication is much younger and still being developed, it is not yet a finished product and performance can only improve as more optimisations are made.

The custom 8-DFE implementations in theory should be 8x faster than the custom 1-DFE implementation. An acceleration of between 6.5x and 7.7x was achieved when moving to an 8-DFE implementation from a 1-DFE implementation so the custom DFE implementation scales well to all the DFEs within one MPC-X node of the Daresbury machine. If (unrealistic) perfect scaling to eight DFEs is assumed, the Maxeler App matrix multiply has accelerations ranging between 256x and 296x over serial C. This is more comparable to the CUBLAS performance.

Table 5.1: Total resource usage for custom DFE implementation

LUTs	**FFs**	**BRAMs**	**DSPs**	
524800	1049600	2567	1963	**total available FPGA resources**
136895	197074	1120	100	**total resources used**
26.0%	18.8%	43.6%	5.1%	**% of available**

5.5 Limitations of custom design

The reason for the custom DFE application not achieving the MaxApp application performance is that the hardware resources; the lookup tables (LUTs), flip-flops (FFs), BRAMs (Block RAM) and multipliers (DSPs), used for the design are not fully utilized as shown in Table 5.1. The custom DFE design is a naive matrix-matrix multiply and it is a non-trivial task to effectively achieve high resource usage. Not all of the DSP blocks are being used to perform 1963 multiplications per clock tick, the theoretical maximum. Only 5.09% are being used, with 96 DSP blocks used for multiplication as this is the burst size of the LMem. One way to improve on this would be to replicate the kernel graph, having a second stream from LMem, the same cyclic computation operating on this second stream in parallel to the first stream and another stream outputting the result to LMem so the second BChunk is processed simultaneously within the kernel, rather than requiring the kernel to run twice. Unfortunately, this is the limit with this idea, there can be at most 2 streams from the LMem as already nearly half of the available BRAM is used with just one stream.

As can be seen by the acceleration results, the Maxeler App outperforms a custom DFE implementation by approximately 10x. The custom design is a naive implementation whereas the Maxeler App implementation has been created by experienced dataflow engineers from Maxeler. As such the Maxeler App design makes better use of the available resources and compute capability of the DFE as can be seen in Table 5.2. The percentage of DSP blocks which are used for multiplication has increased from 5% to nearly 80% whilst the other resources used have doubled. More multiplications are being done in parallel on chip. The design is running at a

Table 5.2: Total resource usage for MaxApp implementation

LUTs	FFs	BRAMs	DSPs	
524800	1049600	2567	1963	**total available FPGA resources**
287815	405977	2567	1538	**total resources used**
54.8%	38.7%	100.0%	78.4%	**% of available**

frequency of 150 MHz but it is likely the design could run at 200 MHz [188] so the performance of Maxeler App could be improved even further. Additionally, the app does the accumulation of the partial results computed of the matrix C on the CPU currently. It would increase performance to use LMem to store the intermediate results and perform the accumulation on the DFE and these changes may be made to the app in the future.

5.6 MaxApp implementation with maximum compute efficiency

For a DFE compute efficiency of 1, where the matrix size is a multiple of the tile size and no padding is required, the acceleration results are shown in Fig. 5.7 for the Maxeler App DFE implementation provided by Maxeler, a CUBLAS implementation and a 1-thread MKL implementation. A matrix dimension N = 384 = tile size was chosen as a starting point, increasing up to N = 9984 = 26*tile size.

When the compute efficiency is maximum, the performance of the Maxeler App DFE implementation has greatly improved to an acceleration of 250x over a serial C implementation, compared to 30-40x from the previous problem dimensions where the matrices needed to be padded to multiples of the tile size. The Maxeler App DFE implementation outperforms the 1-thread MKL with a runtime slightly faster than previously with 5-6x acceleration. However, the CUBLAS implementation, running on a top-of-the-range K40 GPU card, with an acceleration increasing to 2500x still far outperforms the Maxeler App DFE implementation running on an equivalent top-of-the-range MPC-X dataflow solution, with at best a 2000x acceleration if all

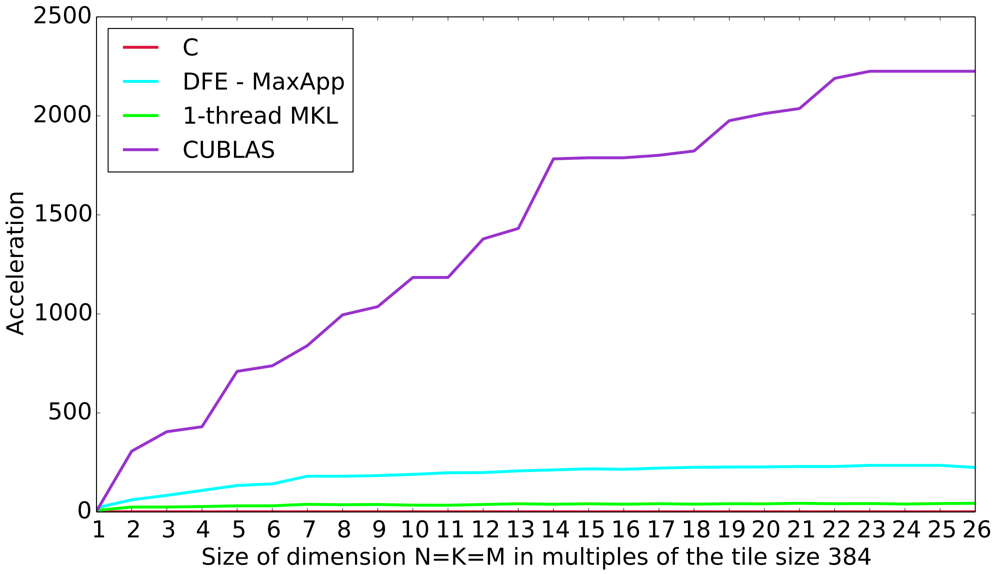


Figure 5.7: The acceleration over serial C of square matrix multiplication ranging from N = 384 to N = 9984 for Maxeler DFE implementation, CUBLAS and 1-thread MKL.

eight of the DFEs within the MPC-X are fully utilized. However, this may not be a fair comparison in terms of resources as the K40 GPU only has 12 GB memory and the MPC-X has 384 GB, 48GB for each DFE in the node. Additionally, the cost of the GPU is likely to be significantly less than the unlisted price for an MPC-X node due in part to the wider user base for GPU technology.

The Maxeler dense matrix multiplication app has a significantly reduced development time, compared to the self-developed implementation which is highly attractive to users in the scientific community. In addition, it provides a much greater acceleration. However, the Maxeler solution still provides relatively poor acceleration when compared to the CUDA CUBLAS implementation and the development time is roughly equal in each case. Where the Maxeler solution may compete with the GPU solution is in terms of energy efficiency, and this is detailed further in the next section.

5.7 Energy efficiency

As application problem sizes have increased in recent years, the running/electricity costs of machines and therefore their energy efficiency has become an important consideration. The annual energy cost to power the top supercomputers exceeded the acquisition cost of the system itself for the first time in 2008 [189]. Further, the target of a 20 MW exaflop system would require a 56x performance improvement with only a 2x increase in power consumption [189]. A 2011 study [190] found that computations per kilowatt-hour has doubled every 1.57 years. Assuming this trend, to get a 56x increase in performance for only a 2x increase in power consumption would take approximately $1.57\log_2 28 = 7.5$ years. We are expected to reach the exascale in 2020 so reaching this performance per kilowatt hour is potentially unachievable without new technology and architectures.

Many studies have been performed which analyse and compare the power consumption and energy efficiency of algorithms running on different devices. To calculate the energy efficiency of the DFE matrix multiply application, the procedure presented in [191] and [192] is followed here, which uses a measure of power consumption in terms of the thermal design power. Here, as in [191], the performance in giga-floating point operations per second (GFLOPS) is analysed to give energy efficiency in terms of GFLOPS per watt, where as in [192], the authors consider energy efficiency in terms of Joules. The number of GFLOP's required to carry out a matrix multiplication for square matrices with dimension N is $2 \times 10^{-9} \times N^3$. The GFLOPS for each device and size of matrix can then be calculated based on the runtime of the application. The power consumption figures used are those stated as the thermal design power of the E5-2670 CPU, the average board power of the K40C and the power figure per DFE card reported by the Maxeler `maxtop` command line utility.

Using these figures for the power consumption is slightly misleading as this is the theoretical maximum power drawn by each device. This assumes that all of the power drawn is used in the calculation where as in reality there will be background tasks and other operations being performed. However, in the absence of more accurate techniques to measure the power consumed, the thermal design power figures

Table 5.3: Power efficiency

	GFLOPS	Acceleration	GFLOPS/watt efficiency	Normalised efficiency
CPU	0.5	1.0x	$4.4 * 10^{-3}$	1.0x
MKL	19.5	38.0x	0.2	38.2x
DFE	107.7	211.0x	5.2	1168.8x
CUBLAS	1057.1	2072.0x	4.5	1015.4x

were used as in [191] and [192]. It should be noted that to get a more accurate figure for the power consumption, an Olson remote power monitoring meter [193] could have been used as in [194]. This would be done by measuring the idle power consumption and the power during the application run and taking the difference. However, this would require this equipment which was not readily available, and also machine access which is impractical for the machines used in this work which are located at Daresbury and Culham. Alternatively, one could use a library like Performance Application Programming Interface (PAPI) [195] or Accurately Measuring Power and Energy for Heterogeneous Resource Environments (Ampehre) [196]. These were investigated but proved to be difficult to install and use on the required machines.

Fig. 5.8 shows the performance per watt for the serial CPU, DFE, CUBLAS and 1-thread MKL implementations. We see the DFE implementation has the best performance per watt due to its low power consumption. Whilst the CUBLAS implementation appears to have fairly good performance per watt, this is mostly due to its excellent performance, rather then its low power consumption. The board power required by the GPU is an order of magnitude higher than the DFE. Table 5.3 shows the typical values for the performance per watt and power efficiency for the different implementations with matrix dimension N = 9984.

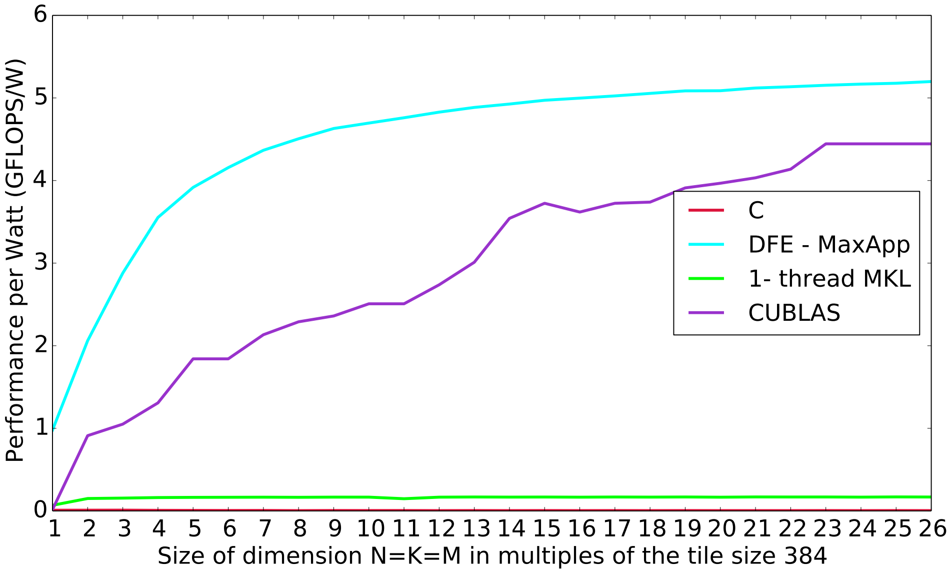


Figure 5.8: The performance per watt (GFLOPS/watt) for the serial C, DFE, CUBLAS and 1-thread MKL implementations for square matrix multiplication ranging from N = 384 to N = 9984.

5.8 Final words on dataflow programming

The dataflow approach provides a significant acceleration of a massive matrix-matrix multiplication and performs on a par with its competitors whilst being much more energy efficient which is now an important factor in choice of technology. Whilst the DFE shows promise, there is still a large hurdle to overcome for those not experienced in FPGA programming to enable the DFE to become more widely adopted in the scientific community. However it is clear that the potential is there for the DFE to perform on a par with the GPU in terms of a massive matrix multiplication once the available DFE resources are fully exploited. Indeed, the Himeno benchmark, a floating point computation kernel known to be bound by memory bandwidth, performs better on a DFE than on a GPU [197]. This suggests that memory bound applications, such as matrix multiplication, could potentially have better performance on a DFE compared to a GPU (once the the implementation has been fully optimized) as the Himeno benchmark is a worst case scenario for memory bound applications.

Programming the DFE is a lengthy process and complicated in comparison to using a GPU or Intel MKL and this programmer effort should be taken into account when considering which accelerator to port a code to. Fortunately, Maxeler technologies are developing an app gallery with DFE implementations of many common problems which can be used by the scientific community. This drastically reduces the development cycle for end users and makes the DFE a more attractive option to non-specialists. The Maxeler dense matrix multiplication app has a significantly reduced development time, compared to the custom implementation and provides a greater acceleration.

This chapter has highlighted the benefit of accelerators in computationally expensive problems. The GPU application provides huge speed-ups compared to a serial and multi-threaded CPU implementation, reducing the run-time significantly and the Maxeler App DFE implementation performs well and shows promise. Whilst the GPU is at this time more accessible to the general scientific community with highly optimized libraries provided such as CUFFT and CUBLAS, meaning the highest performance is achievable with little user development, it is not inconceivable that in the future the DFE will become more accessible. High performance libraries like MaxBLAS [198] will be developed and the machine may outperform the GPU in the future and become a more attractive option for the scientific community. Further, as running costs become an important consideration, the DFE approach holds an advantage due to the superior energy efficiency it provides. The DFE implementation has a high performance per watt ratio and low power consumption since it has a low clock frequency, making it an ideal choice of technology for energy efficient computing.

Whilst the Maxeler dataflow programming paradigm is quite difficult for those beginning in FPGA programming, there are other tools available on the market which may be easier for beginners such as Vivado High Level Synthesis (HLS) from Xilinx [199] (formally AutoPilot from AutoESL which was acquired by Xilinx in 2011). This is a compiler that compiles C/C++ programs and performs advanced platform based code transformations and synthesis optimizations to produce Verilog or VHDL code for FPGAs [200]. The design time is obviously much shorter as users can develop in C/C++ and not require learning MaxJ, the Java-like language used to program the DFE kernel and manager code as described in Chapter 2. Users have reported tremendous reductions in the necessary effort for designing FPGA accelerated systems [201]. Other HLS tools are available and are described in a 2015 survey [202]. This includes Xilinx Vivado HLS; Altera OpenCL which is a cross-platform language targeting many parallel computing platforms including GPUs and FPGAs and is C-like; Bluespec System Verilog, a high level hardware description language built upon a subset of System Verilog which is somewhere between a HDL and HLS; LegUp which takes C code in it's entirety and compiles it for FPGA and ROOCC, a C to VHDL compiler. LegUp and ROOCC are both research tools where as the others are available commercially. Vivado HLS, LegUp and ROOCC have small learning curves due to them essentially being C to HDL compilers but to get better performance, optimisations must be made to the code. OpenCL has a mid-level accessibility and Bluespec System Verilog has a steep learning curve as it does not accept C code.

Global companies such as IBM and Intel are now also beginning to invest in high performance computing on FPGAs. IBM has the Coherent Accelerator Processor Interface (CAPI) which on POWER8 systems provides a high-performance solution for the implementation of client-specific, computation-heavy algorithms on an FPGA [203]. Intel acquired Altera recently and is aiming to integrate Arria 10 GX FPGA and Broadwell EP chip onto the same die [204] which will reduce power consumption and improve communication between the CPU and FPGA. Xilinx offers the Zynq-7000 family [205] of system on chip (SoC) which features an ARM CPU and FPGA on the same chip providing a smart, intelligent system aiming for the best performance at lowest possible power consumption. It is clear FPGA use for high performance computing is increasing in popularity and there are many tools now on the market to assist in the easy adoption of this architecture by the general scientific community where FPGA use has been limited to experts previously.

Chapter 6

Conclusions

The aim of this thesis has been to investigate the potential of emerging architectures to increase the capabilities of data processing tasks and simulations within the field of fusion. There is a close relationship between advances in the capabilities of plasma physics computations and the recent advances in technology such as the emergence of GPU and FPGA architectures. Three examples from different fusion applications have been studied and these technologies have been used to make a contribution in different areas in plasma physics.

The summary of the field presented in Chapter 1 indicates the trend for larger, more complex simulations requiring more computing power. We have reached an era where to get more computing power for complex plasma physics calculations, alternative accelerator technologies need to be explored further. Traditional CPU-only approaches are no longer sufficient to provide the increased computing power and more flexible architectures are required. The underlying hardware of two of these new technologies, GPUs and FPGAs, have been discussed and the software models used to exploit their parallelism introduced.

In Chapter 3, the application of GPU technology to the high date-rate synthetic aperture microwave imaging (SAMI) diagnostic was investigated. The raw data processing task was accelerated 60x compared to the original IDL data processing code, which then enabled for the first time multi-shot analysis of the SAMI data to be undertaken. In the future, on NSTX-U and MAST-U, the SAMI raw data will now be able to be processed, and therefore analysed, between consecutive shots for the first time, greatly improving the capabilities of SAMI on these machines. The GPU code has been used to process all of the stored raw data from previous MAST campaigns which has led to a consistent log-linear relationship between the ECE power and D_α emission across multiple shots being observed. Several physical reasons for the observed relationship were investigated, attempting to find a model that fully described the data. Currently, there are potentially further considerations that need to be included in the model to accurately describe observed trends.

The potential of using GPUs to accelerate the industry standard FISPACT-II code which calculates the nuclide inventory after neutron irradiation was investigated in Chapter 4, showing promise for a partial or even a full port of the code to the GPU architecture. The GPU code (using a new matrix exponential method based on Chebyshev rational approximations to calculate the inventory) was tested against a simple FISPACT example (which uses the LSODES package, a linear multi-step method, to calculate the inventory) and the calculated nuclide inventories agree to within 0.01%. The chosen MAGMA library routine which calculates the nuclide inventory accurately is however slow, offering no acceleration over the FISPACT calculation. There are other library routines which could be used in place of the MAGMA routine to produce the correct results and provide an acceleration. Specifically, the MAGMA Sparse-Iter package is of interest and can provide an implementation of the current MAGMA routine but optimized for sparse matrices.

A new FPGA-based technology known as the "dataflow engine" was explored in the context of a problem common to not only many simulations found in fusion applications, but generally in the field of computational science. A custom implementation of a large matrix multiplication problem was investigated along with a highly optimised Maxeler App implementation provided by Maxeler Technologies. It was found to be difficult to achieve the same performance as the Maxeler App without extensive specialised code optimisation, but the many off-the-shelf Maxeler App implementations, available for diverse applications, provide quick, easy to use, high performance solutions to dataflow problems. Whilst at this time more established technologies like the GPU provide better performance for this particular problem, the dataflow technology shows promise and certainly outperforms the GPU in terms of energy efficiency, which is now also an important consideration and expense for larger simulations requiring more computing power. Additionally, the DFE outperforms the GPU in certain specialised applications and FPGA technology in general looks set to gain in popularity in light of investment by large global companies in hybrid FPGA-CPU architectures.

For modelling next generation tokamaks like ITER and DEMO, the fusion community needs to be able to run much more sophisticated models to accurately simulate physical processes occurring in these machines. Algorithms requiring massive computing power will be employed and it is not clear, given recent trends, if this can be provided by traditional approaches such as many-core CPU MPI-OMP based solutions. Alternative accelerator technologies may be required to provide the necessary computing power to successfully carry out these demanding simulations.

Once ITER is operational, the amount of diagnostic data produced will be at least an order of magnitude higher than current machines like JET and MAST. Consideration of how to handle these large amounts of data also needs to be done now in order to prepare for ITER. The work on the GPU data processing code for the high data rate SAMI diagnostic presented in Chapter 3 demonstrates the use of emerging technology for handling large amounts of off-line data. The successful implementation of a GPU data processing code will hopefully encourage others to adopt GPU technology, where appropriate, to process the large amount of data expected for all diagnostics on ITER due to the longer pulse length (ITER is aiming for 400-600 second pulses). The SAMI diagnostic has the highest data rate on MAST of 4GB per shot and the data processing with the existing IDL code was too slow for inter-shot processing. Even overnight processing of the collected data for that day did not occur and the raw data was stored for processing at some later date. Storing a large amount of raw data for ITER will not be acceptable so accelerated methods for data processing based on emerging technologies which dramatically reduce the processing time are attractive.

Whilst the FISPACT-II code is the industry standard used to calculate nuclide inventories of materials after neutron irradiation, it is limited to studying one region in a tokamak at a time. Large simulations, modelling many regions in a tokamak can be done, but the nuclide inventory for each region is calculated sequentially in FISPACT and takes a long time. Being able to perform these full, detailed simulations in a sensible amount of time for tokamaks like ITER and DEMO is essential for predicting material degradation under different flux conditions and initial material composition. GPU technology can provide this capability. This work has demonstrated a significantly accelerated cross-section collapse useful for investigating different flux scenarios. Whilst further work needs to be done on the nuclide inventory calculation involving the calculation of the matrix exponential on a GPU, the foundation for a full GPU version of FISPACT has been made. Further sparse libraries such as the MAGMA Sparse Iter library need to be explored for use in the chosen CRAM method to calculate the matrix exponential. Additionally, the potential for using a time stepping method to solve the system of linear equations, instead of the matrix exponential method, could be explored. There have been successful implementations of the Runge-Kutta method on a GPU for example as discussed in Chapter 4.

FPGA-based technologies have been extensively used in data acquisition and signal processing tasks in the fusion community. In Chapter 5 the FPGA has been explored as a key component of a dataflow engine (DFE) for high performance computing. This could include large data-processing tasks or large-scale simulations relevant for ITER. Currently the DFE technology is not as easy to use as the GPU technology and so requires a greater programmer effort and development time. Additionally, most (but not all) GPU applications still outperform applications run on the DFE due to the extensive number of highly optimized libraries and, more generally, the relative maturity of the GPU technology compared to the DFE technology. FPGA technology in general is important to study now however, as its use is likely to increase in the future as large global companies such as IBM and Intel invest further in the field of high performance computing on FPGAs.

Overall, this work has achieved its primary aim of investigating the use of emerging architectures applied to problems faced by the fusion community, and shown it is beneficial to do so in a wide range of fusion applications including the processing of large amounts of raw data from fusion diagnostics and large scale fusion relevant simulations. Investigating the use of emerging high performance computing architectures will continue to be essential to the fusion community for the planning and operation of next generation devices such as ITER and beyond.

Appendix A

SAMI data-processing CUDA kernels

A.1 Data-conditioning kernels

```
//Calculate the low frequency switching noise and remove it
//Kernel to smooth an array A of size N*M (2D array with N rows, M columns) with a boxcar average of the specified width w. Details here http://www.exelisvis.com/docs/SMOOTH.html
__global__ void smooth(float *A, float *R, int dn, int Nant, int n_sweeps, int nf, int w){

  int bid = dn*blockIdx.x;
  int tid;

  float R0, w1;
  int l;            // dummy variables
  if(w % 2 == 0){
    w++;            // "If a Width value is even, then Width+1 will be used instead."
  }
  w1 = 1.0f / (float)(w);

  int size = dn*Nant*n_sweeps*nf;

```

```
//The algorithm given in link above, modified to ensure chunk[smthItt,*] -= smooth(chunk[smthItt,*],smth).
  while(bid<size){
    tid = threadIdx.x;
    while(tid<dn){
      if(tid>=(w-1)/2 && tid<=dn - (w+1)/2){
        R0 = 0.0f;
        l=0;
        while(l<w){
          R0 += A[tid + bid - (w-1)/2 + l];
          l+=1;
        }
        R[tid + bid] = A[tid + bid] - w1*R0;     //to account for -=
      }
      tid += blockDim.x;
    }
    bid+=dn*gridDim.x;
  }
}

//Take only the middle part of the data to avoid any residual switching noise when switching between frequency channels to extract noise from the beginning and end of each signal
__global__ void middle(float *A, float *R, int dn, int nInt, int Nant, int n_sweeps, int nf, int n_delay, int incr){

  int bid = nInt*blockIdx.x;
  int obid = dn*blockIdx.x;
  int tid;

  int size = nInt*Nant*n_sweeps*nf;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      R[tid + bid] = A[tid + obid + n_delay - incr];
      tid += blockDim.x;
    }
    bid+=nInt*gridDim.x;
    obid+=dn*gridDim.x;
  }
}

//This shifts the correct channels to compensate for the ADC timing errors on data taken with firmware built on 23/09/2011.
//Kernel to shift data in specified column (applied to channels 0, 1, 8 and 9)
__global__ void shift(float *A, float *R, int nInt, int Nant, int n_sweeps, int nf, int column){

  int bid = Nant*nInt*blockIdx.x;
  int tid;
  float temp;

  int size = nInt*Nant*n_sweeps*nf;

  while(bid<size){
    tid = threadIdx.x;
    temp = A[bid + column*nInt];
    while(tid<nInt - 1){
      R[bid + column*nInt + tid] = A[bid + column*nInt + tid + 1];
      tid += blockDim.x;
    }
    R[bid + column*nInt + nInt - 1] = temp;
    bid += Nant*nInt*gridDim.x;
  }
}

//Kernel to recombine shifted data in specified column with full data array to correct for ADC timing errors (applied to channels 0, 1, 8 and 9)
__global__ void combine(float *A, float *R, int nInt, int Nant,
    int n_sweeps, int nf, int column){

  int bid = Nant*nInt*blockIdx.x;
  int tid;
  int size = nInt*Nant*n_sweeps*nf;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      R[bid + column*nInt + tid] = A[bid + column*nInt + tid];
      tid += blockDim.x;
    }
    bid += Nant*nInt*gridDim.x;
  }
}
```

A.2 First Fourier filter for I and Q components

```
//Kernel to make data cufft complex for the cufft's, imaginary part zero
__global__ void make_complex(int Nant, int nf, int nInt, int nSec, float *dev_data, cufftComplex *dev_complex_data){

  int bid = blockIdx.x*blockDim.x + threadIdx.x;
  int size = nInt*Nant*nSec*nf;

  while(bid<size){
    dev_complex_data[bid].x = dev_data[bid];
    dev_complex_data[bid].y = 0.0f;

    bid += blockDim.x*gridDim.x;
  }
}

//Kernel to apply filter function on even columns
__global__ void mult_three(double a, int nInt, int Nant, int nSec, int nfSelect, cufftComplex *b, cufftComplex *c, cufftComplex
    *r){

  cuFloatComplex newa = make_cuFloatComplex(a, 0.0f);

  int bid = 2*nInt*blockIdx.x;
  int tid;
  int size = 2*nInt*Nant*nSec*nfSelect;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      r[bid + tid] = cuCmulf(newa, cuCmulf(b[bid + tid], c[tid]));
      tid += blockDim.x;
    }
    bid += 2*nInt*gridDim.x;
  }

}

//Kernel to apply filter function on odd columns and to apply calibration data correctiing for the phase drift between I and Q components caused by cable lengths and filters as a function of IF
__global__ void complex_mult(double a, int nInt, int Nant, int nSec, int nfSelect, float *dev_phase, cufftComplex *b, cufftComplex *c, cufftComplex *r, int column ){

  cuFloatComplex newa = make_cuFloatComplex(a, 0.0f);

  int bid = nInt + 2*Nant*nInt*blockIdx.x;
  int tid;
  int size = 2*nInt*Nant*nSec*nfSelect;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      r[bid + column*nInt + tid] = cuCmulf(newa, cuCmulf(b[bid + column*nInt + tid], cuCmulf(c[tid], make_cuFloatComplex(cos(dev_phase[tid]), sin(dev_phase[tid])))));
      tid += blockDim.x;
    }
    bid += 2*Nant*nInt*gridDim.x;
  }
}
```

A.3 Sideband separation

```
//16 real signals get converted to 8 complex signals for upper and lower sideband and apply calibration data to correct for phase differences between antennaas due to RF electrical lengths
__global__ void complexify(float *a, float *c, float *d, int nInt, int Nant, int nSec, int nfSelect, cufftComplex *data, cufftComplex *dev_complex_data, int column, int index){

  int bid = 2*Nant*nInt*blockIdx.x;
  int cbid = Nant*nInt*blockIdx.x;
  int obid = nInt + 2*Nant*nInt*blockIdx.x;
  int tid;
  int i = 1;

  while(index<Nant*nfSelect){
    while(bid < i*2*Nant*nInt*nSec){
      tid = threadIdx.x;
      while(tid < nInt){
        dev_complex_data[cbid + column*nInt + tid] = make_cuFloatComplex(a[index]*data[bid + 2*column*nInt + tid].x, -c[index]*data[bid + 2*column*nInt + tid].x - d[index]*data[obid + 2*column*nInt + tid].x);
        tid += blockDim.x;
      }
      bid += 2*Nant*nInt*gridDim.x;
      obid += 2*Nant*nInt*gridDim.x;
      cbid += Nant*nInt*gridDim.x;
    }
    index += Nant;
    i += 1;
  }
}

//Kernel to make the  2 dummy matrices, one for upper and one for lower sideband
__global__ void makeDummy(int nInt, int Nant, int nSec, int nf, cufftComplex *dev_complex_data, cufftComplex *dummy){

  int bid = threadIdx.x + blockIdx.x*blockDim.x;
  int size = nInt*Nant*nSec*nf;

  while(bid < size){
    dummy[bid].x = cuCrealf(dev_complex_data[bid]);
    dummy[bid].y = cuCimagf(dev_complex_data[bid]);
    bid += blockDim.x*gridDim.x;
  }
}
```

A.4 Second Fourier filter

```
//Kernel to apply filter function to 2 complex matrices
__global__ void mult_two(double a, int nInt, int Nant, int nSec, int nfSelect, cufftComplex *b, cufftComplex *c, cufftComplex *r){

  cuFloatComplex newa = make_cuFloatComplex(a, 0.0f);

  int bid = nInt*blockIdx.x;
  int tid;
  int size = nInt*Nant*nSec*nfSelect;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      r[bid + tid] = cuCmulf(newa, cuCmulf(b[bid + tid], c[tid]));
      tid += blockDim.x;
    }
    bid += nInt*gridDim.x;

  }

}

//Kernel to apply calibration data correcting for phase offsets and balancing amplitudes between I and Q components via matrix inversion. Differences between I and Q caused by hybrid couplers and amplifiers.
__global__ void phi(int nInt, int Nant, int nSec, int nfSelect, cufftComplex *data, float *phi, int column, int index){

  int bid = Nant*nInt*blockIdx.x;
  int tid;
  int i = 1;

  while(index<Nant*nfSelect){
    while(bid < i*Nant*nInt*nSec){
      tid = threadIdx.x;
      while(tid < nInt){
        data[bid + tid + column*nInt] = cuCmulf(data[bid + tid + column*nInt], make_cuFloatComplex(cosf(phi[index]), sinf(phi[index])));
        tid += blockDim.x;
      }
      bid += Nant*nInt*gridDim.x;
    }
    index += Nant;
    i += 1;
  }
}
```

A.5 Cross-correlation calculation

```
//Kernel to calculate the mean of each group of nInt points
__global__ void complex_mean(double divide, int nInt, int Nant, int nSec, int nf, cufftComplex *complex_matrix, cufftComplex *dev_mean_matrix){

  int bid = nInt*blockIdx.x;
  int nbid = blockIdx.x;
  int tid;
  int size = nInt*Nant*nSec*nf;

  int cacheIndex = threadIdx.x;
  __shared__ cuFloatComplex matrix[256];

  cuFloatComplex new_divide = make_cuFloatComplex(divide, 0.0f);

  while(bid<size){
    cuFloatComplex mean_matrix = make_cuFloatComplex(0.0f, 0.0f);
    tid = threadIdx.x;
    while(tid<nInt){
      mean_matrix = cuCaddf(mean_matrix, complex_matrix[bid + tid]);
      tid += blockDim.x;
    }

    matrix[cacheIndex] = mean_matrix;
    __syncthreads();

    int i = blockDim.x/2; //threads per block must be a power of 2
    while(i!=0){
      if(cacheIndex < i){
        matrix[cacheIndex] = cuCaddf(matrix[cacheIndex], matrix[cacheIndex + i]);
      }
      __syncthreads();
      i /= 2;
    }
    dev_mean_matrix[nbid] = cuCmulf(new_divide, matrix[0]);
    nbid += gridDim.x;
    bid += nInt*gridDim.x;
  }
}

//Kernel to calculate each data element minus it's corresponding mean, and the complex conjugate of that
__global__ void complex_minus(int nInt, int Nant, int nSec, int nf, cufftComplex *dev_mean_matrix, cufftComplex *complex_matrix, cufftComplex *dev_complex_conj_matrix){

  int bid = nInt*blockIdx.x;
  int tid;
  int mid = blockIdx.x;
  int newidx = threadIdx.x + blockIdx.x*blockDim.x;
  int size = nInt*Nant*nSec*nf;

  while(bid<size){
    tid = threadIdx.x;
    while(tid<nInt){
      complex_matrix[bid + tid] = cuCsubf(complex_matrix[bid + tid], dev_mean_matrix[mid]);
      tid += blockDim.x;
    }
    bid += nInt*gridDim.x;
    mid += gridDim.x;
  }


  while(newidx<size){
    dev_complex_conj_matrix[newidx] = cuConjf(complex_matrix[newidx]);
    newidx += blockDim.x*gridDim.x;
  }
```

```
}

//Kernel to calculate cross-correlations for each antenna pair, frequency sweep and upper and lower sideband
__global__ void cross_correlations(double divide_minus_one, int nInt, int Nant, int nSec, int nf, cufftComplex *complex_matrix, cufftComplex *dev_complex_conj_matrix, cufftComplex *final){

  int bid;
  int obid;
  int tbid = nInt*Nant*blockIdx.x;
  int tid;
  int j, m;
  int l = 0;

  int cacheIndex = threadIdx.x;
  __shared__ cuFloatComplex matrix[256];

  float divide = 1.0f/nInt;

  cuFloatComplex new_divide_minus_one =  make_cuFloatComplex(divide_minus_one, 0.0f);

  while(l<nf){
    while(tbid<(l+1)*nInt*Nant*nSec){
      bid = 0;
      m = 0;
      while(bid<nInt*Nant){
        obid = 0;
        j=0;
        while(obid<Nant*nInt){
          cuFloatComplex thread_sum = make_cuFloatComplex(0.0f, 0.0f);
          tid = threadIdx.x;
          while(tid<nInt){
            thread_sum = cuCaddf(thread_sum, cuCmulf(complex_matrix[tbid + bid + tid], 
               dev_complex_conj_matrix[tbid + obid + tid]));
            tid += blockDim.x;
          }

          matrix[cacheIndex] = thread_sum;
          __syncthreads();

          int i = blockDim.x/2;
          while(i!=0){
            if(cacheIndex < i){
              matrix[cacheIndex] = cuCaddf(matrix[cacheIndex], ↩
                 matrix[cacheIndex + i]);
            }
            __syncthreads();
            i /= 2;
          }
          final[m + Nant*j + (int)(divide*Nant*nf*tbid) - l*nSec*↩
             Nant*Nant*nf + l*Nant*Nant] = cuCmulf(↩
             new_divide_minus_one, matrix[0]);
          obid += nInt;
          j += 1;
        }
        m += 1;
        bid += nInt;
      }
      tbid += nInt*Nant*gridDim.x;
    }
    l += 1;

  }
}
```

A.6 Temporal profiles for the remaining shots

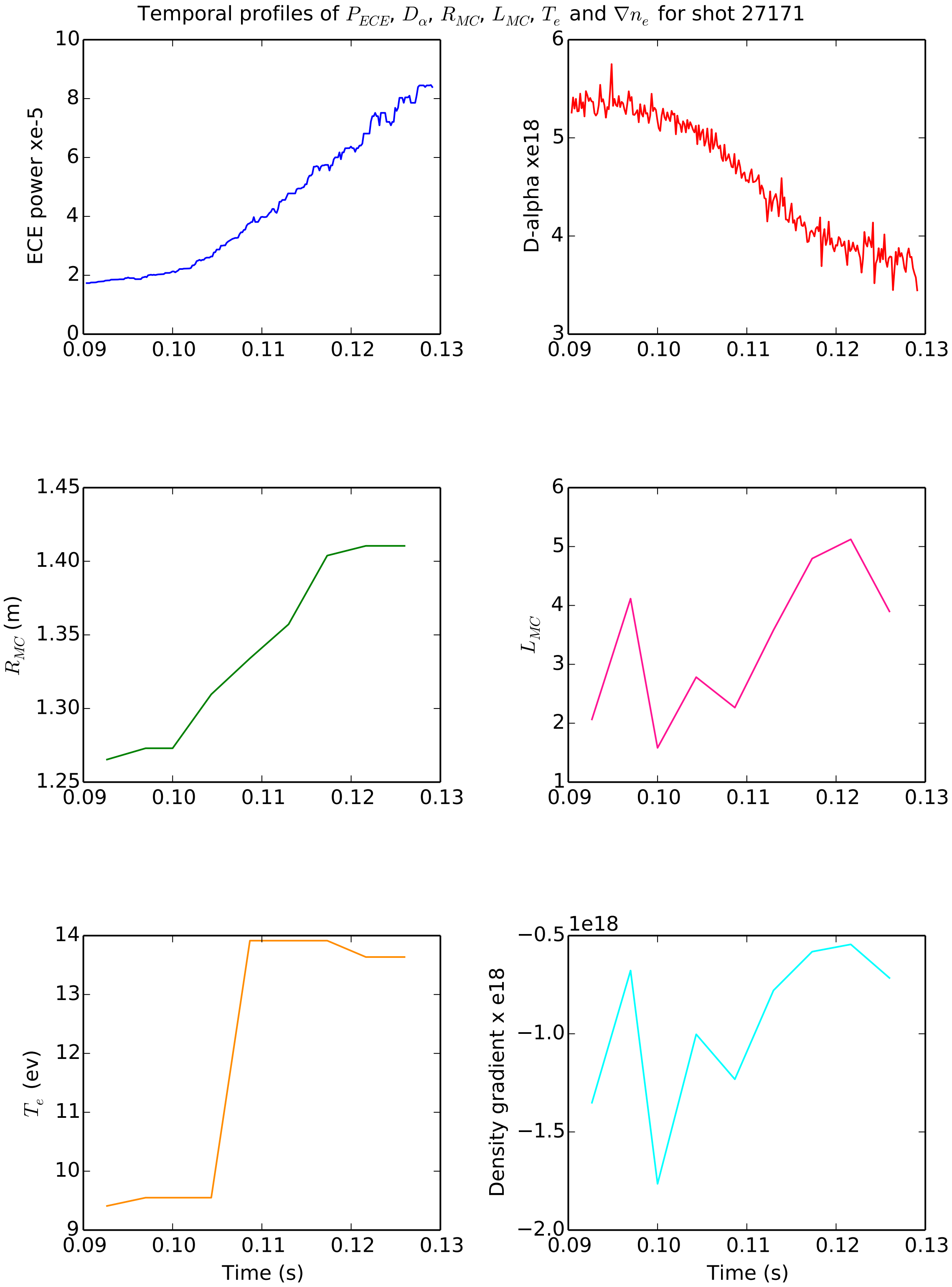


Figure A.1: Temporal profiles of ECE power (top left), D$_\alpha$ emission (top right), radial location of MC window (middle left), density scale length (middle right), electron temperature at MC window (bottom left) and density gradient (bottom right) for shot 27171.

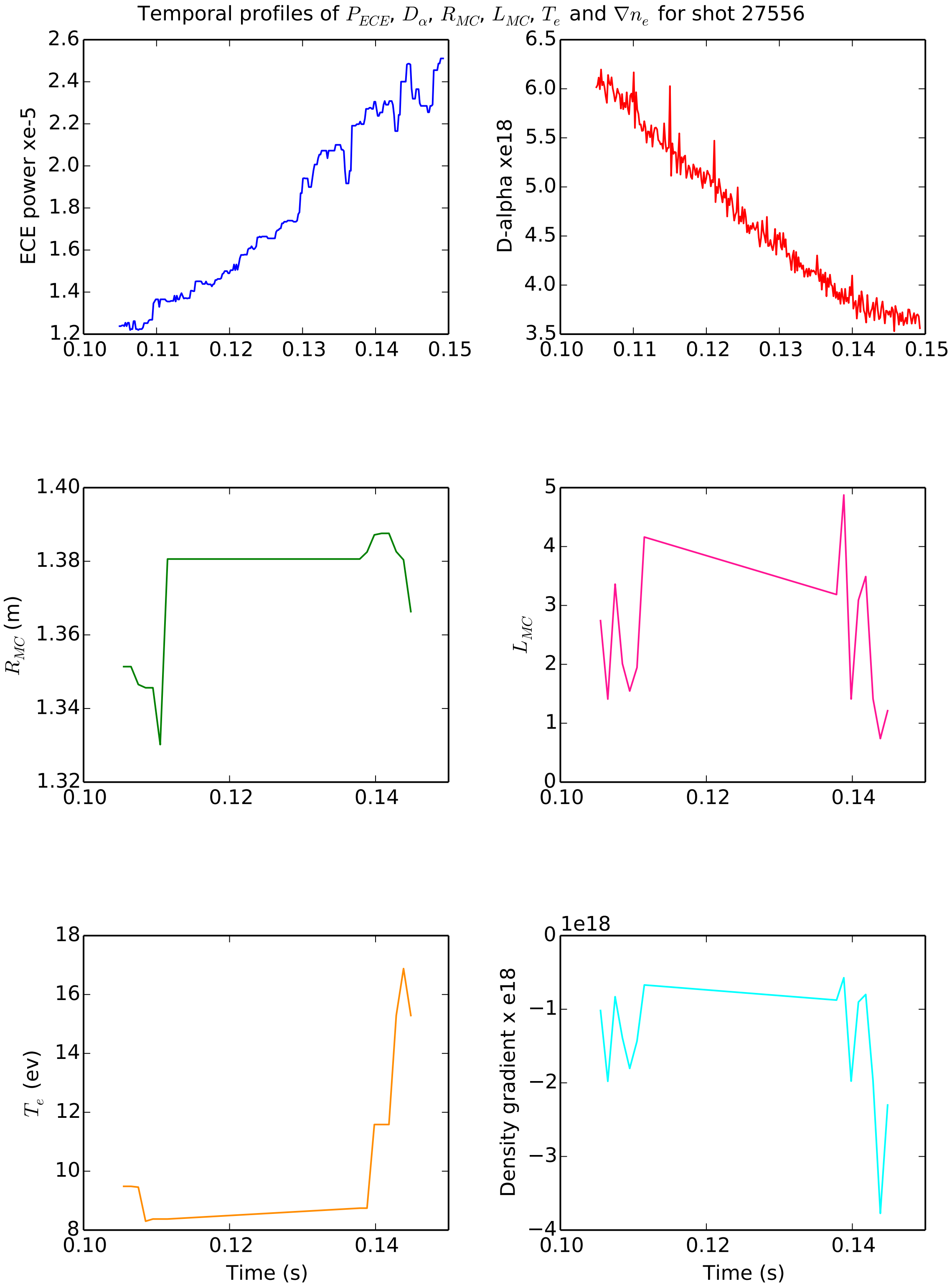


Figure A.2: Temporal profiles of ECE power, D_α emission, radial location of MC window, density scale length, electron temperature at MC window and density gradient for shot 27556.

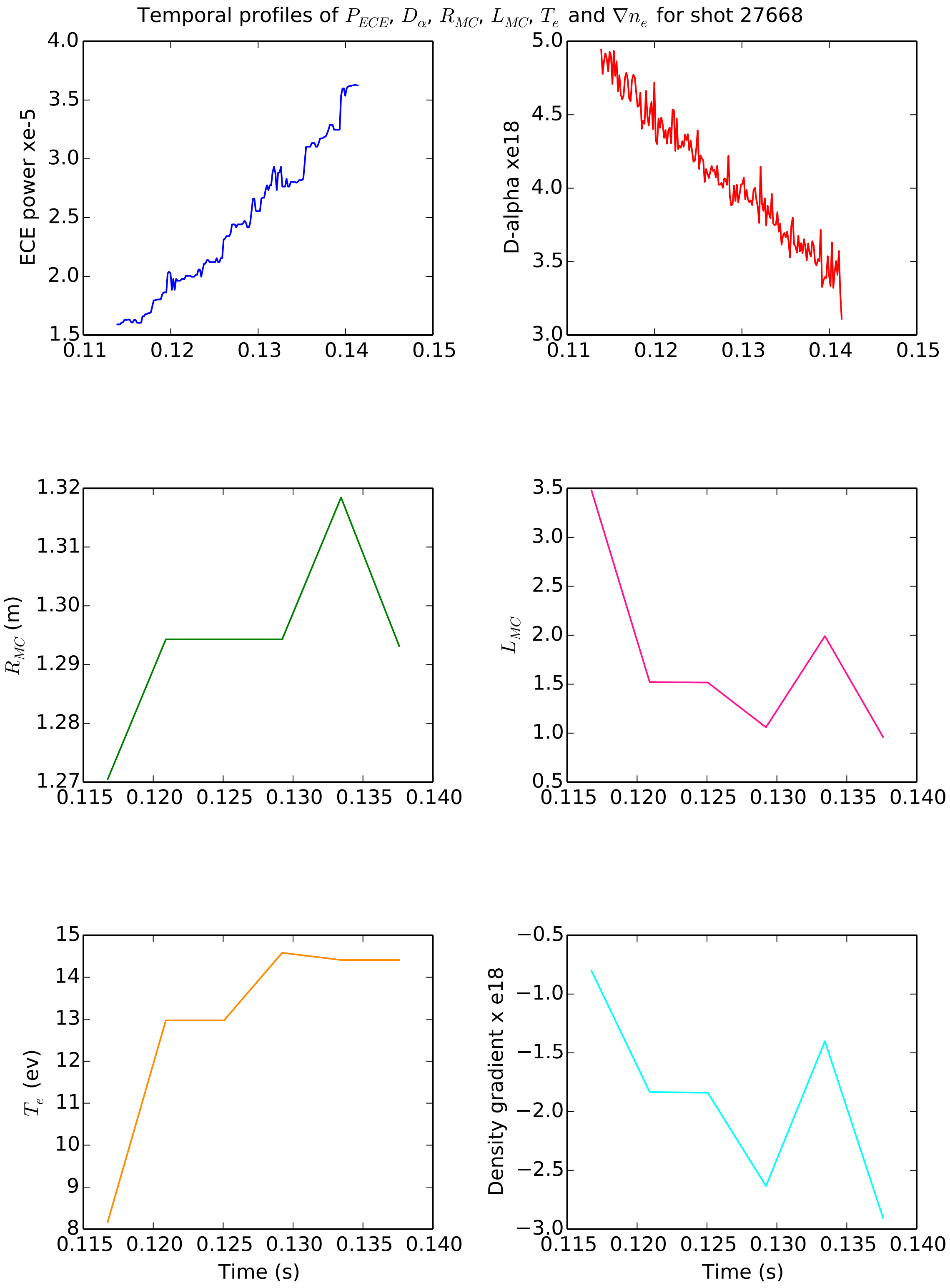


Figure A.3: Temporal profiles of ECE power, D$_\alpha$ emission, radial location of MC window, density scale length, electron temperature at MC window and density gradient for shot 27668.

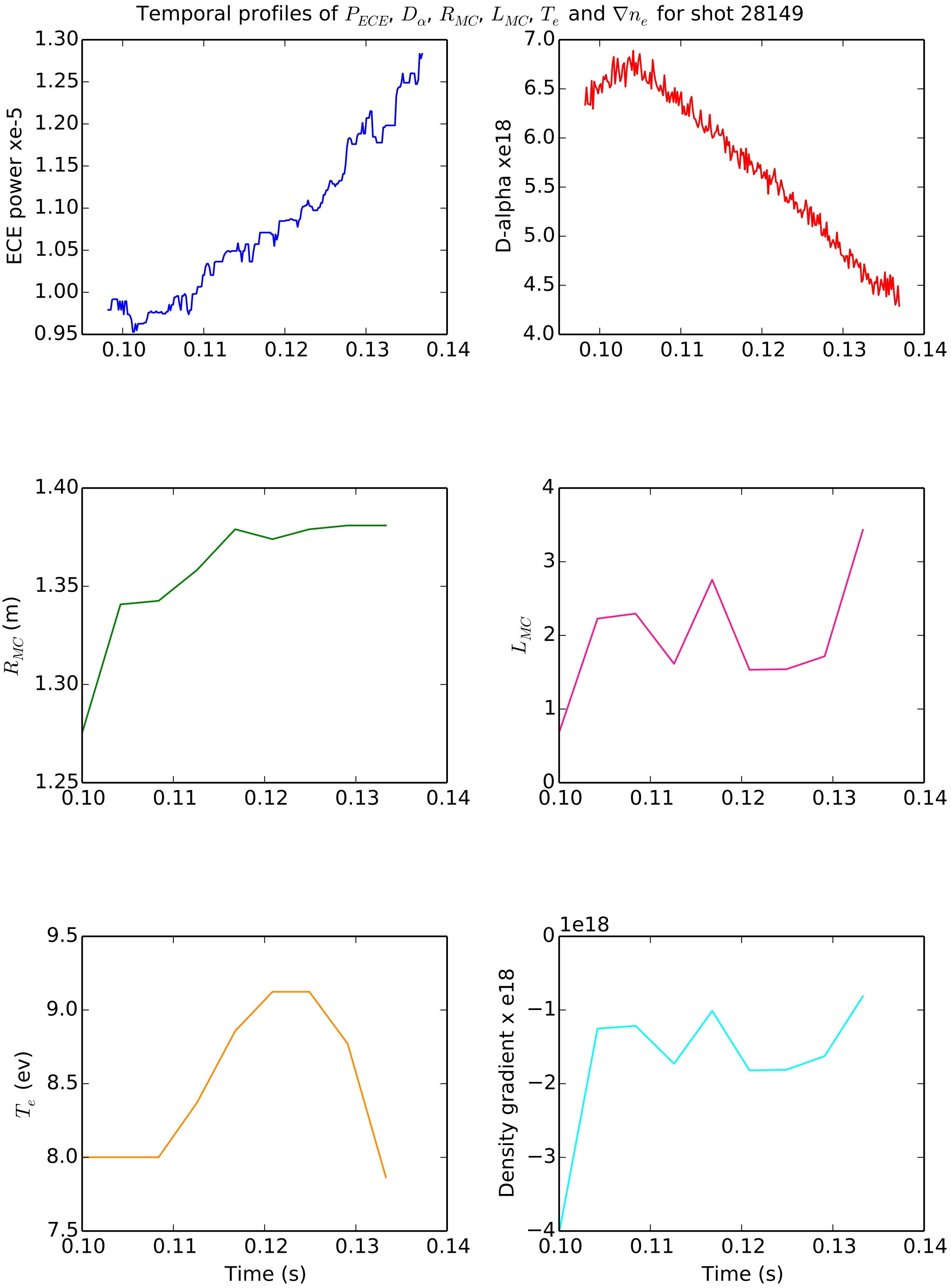


Figure A.4: Temporal profiles of ECE power, D$_\alpha$ emission, radial location of MC window, density scale length, electron temperature at MC window and density gradient for shot 28149.

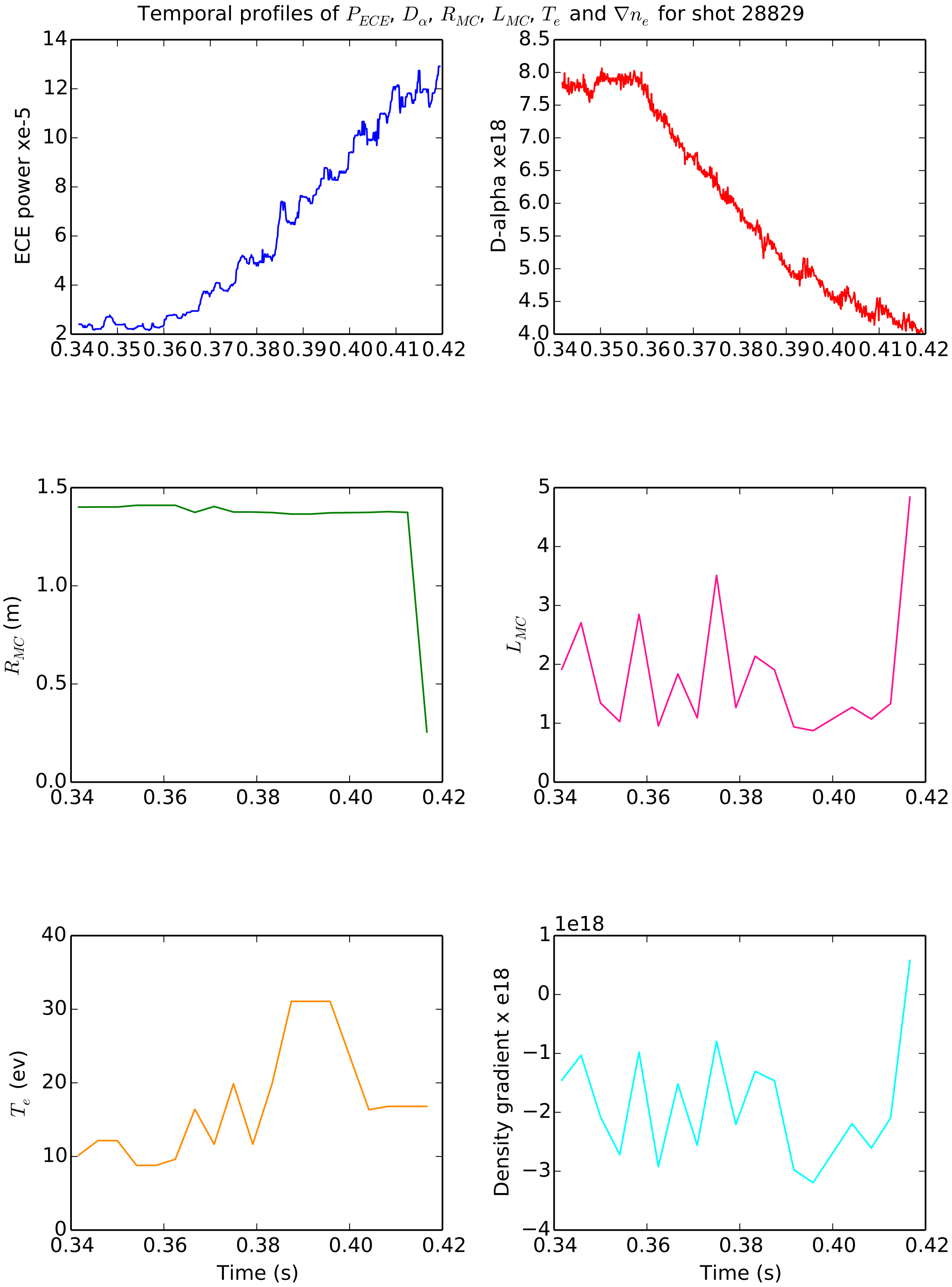


Figure A.5: Temporal profiles of ECE power, D$_\alpha$ emission, radial location of MC window, density scale length, electron temperature at MC window and density gradient for shot 28829.

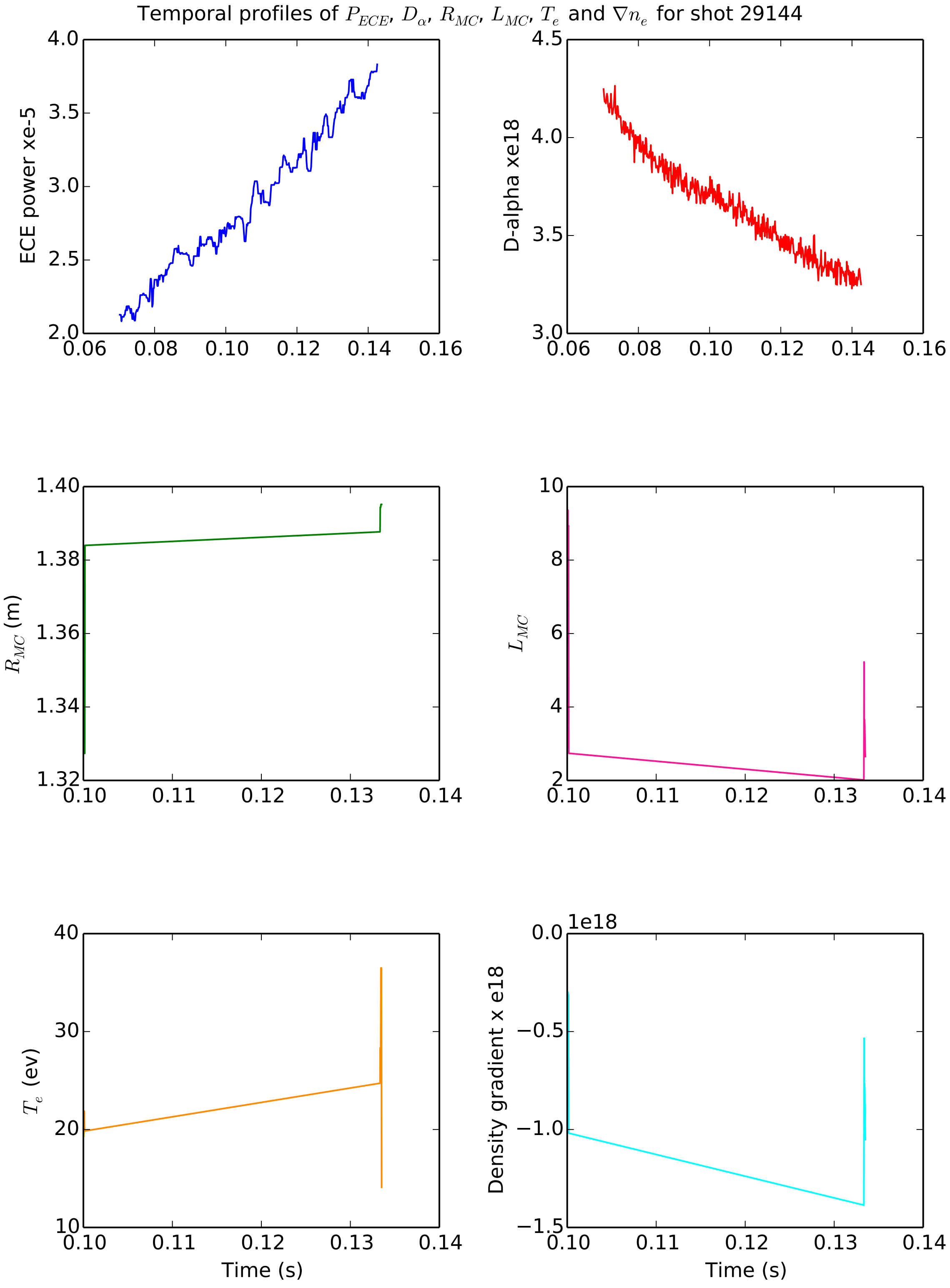


Figure A.6: Temporal profiles for shot 29144. All the previous shots show similar behaviour: the ECE increases, the D_α decreases and T_e increases, as the plasma is moving out. Not much information can be obtained from L_{MC}, which depends on the density gradient.

Appendix B

GPU code for Bateman solver

B.1 Cross-section collapse

```
//Take the reciprocal of the flux column sums
__global__ void reciprocal(double* vector, int Cells){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<Cells){
    vector[tid] = 1.0/vector[tid];
    tid += blockDim.x*gridDim.x;
  }
}

//Divide each element in a column by the sum of the flux for that column
__global__ void normalise(double *output, double *vector, int Iso, int Cells){

  int tid;
  int bid = Cells*blockIdx.x;

  while(bid<Iso*Cells){
    tid = threadIdx.x;
    while(tid<Cells){
      output[bid + tid] = output[bid + tid]*vector[tid];
      tid += blockDim.x;
    }
    bid += Cells*gridDim.x;
  }
}

int main{

  //Initial set up of data
  //...

  //Matrix multiply
  ret = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, M, N, K, &alpha, flux_array_cublas_dev, M, xc_array_cublas_dev, K, &beta, out_array_cublas_dev, M);

  //Column sum of flux matrix (equivalent to matrix vector multiplication with vector of ones)
  retColSum = cublasDgemv(handleColSum, CUBLAS_OP_N, M, K, &alpha, flux_array_cublas_dev, M, vector_in_dev, 1, &beta, vector_out_dev, 1);

  //Normalise result matrix by dividing by entries in each column by the sum of the flux for that column.
  reciprocal<<<32, 256>>>(vector_out_dev, M);
  normalise<<<32, 256>>>(out_array_cublas_dev, vector_out_dev, N, M);

  //...

}
```

B.2 CRAM on the GPU

```
//n_0 = 0
__global__ void zero(cuDoubleComplex* vector, int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    vector[tid] = make_cuDoubleComplex(0.0, 0.0);;
    tid += blockDim.x*gridDim.x;
  }
}

//A = A*t
__global__ void time(double* At, double* A, double t, int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    At[tid] = A[tid]*t;
    tid += blockDim.x*gridDim.x;
  }
}

//A = A - theta(j)*eye(size(A)))  (eye is identity matrix)
__global__ void thetaOp(cuDoubleComplex *newA, double *At, double theta_r, double theta_i, double *identity, int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    newA[tid] = make_cuDoubleComplex(At[tid] - theta_r*identity[tid], -theta_i*identity[tid]);
    tid += blockDim.x*gridDim.x;
  }
}

//n_0 = alpha(j)*n_0
__global__ void alphaOp(cuDoubleComplex *newn_0, double alpha_r, double alpha_i, double *n_0, int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    newn_0[tid] = make_cuDoubleComplex(alpha_r*n_0[tid], alpha_i*↩
        n_0[tid]);
    tid += blockDim.x*gridDim.x;
  }
}

//n = n + (A - theta(j)*eye(size(A))) \ (alpha(j)*n_0);
__global__ void Add(cuDoubleComplex *nd, cuDoubleComplex *newn_0,↩
     int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    nd[tid] = cuCadd(nd[tid], newn_0[tid]);
    tid += blockDim.x*gridDim.x;
  }
}

//n = 2*real(n);
__global__ void Double(double *n, cuDoubleComplex *nd, int size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    n[tid] = 2*cuCreal(nd[tid]);
    tid += blockDim.x*gridDim.x;
  }
}

//n = n + alpha_0*n_0;
__global__ void getN(double *n, double alpha_0, double *n_0, int ↩
    size){

  int tid = threadIdx.x + blockIdx.x*blockDim.x;

  while(tid<size){
    n[tid] = n[tid] + alpha_0*n_0[tid];
    tid += blockDim.x*gridDim.x;
  }
}

int main{

  //Initial data set up (theta, alpha etc.)

  for(nitts=0; nitts<M; nitts+=ngpus){

    //load each gpu with a different activation matrix
    for(l=0; l<ngpus; l++){
      gpuErrchk(cudaSetDevice(l));
      gpuErrchk(cudaMemcpyAsync(A_d[l], HostMatrix + l*rows*cols + nitts*cols*rows, sizeof(double)*rows*cols, cudaMemcpyHostToDevice, streams[l]));

    }

    for(secs=0; secs<timesteps; secs++){

      //A = A*t
      for(l=0; l<ngpus; l++){
        gpuErrchk(cudaSetDevice(l));
        zero<<<32, 256, 0, streams[l]>>>(nd_d[l], rows);
        time<<<32, 256, 0, streams[l]>>>(At_d[l], A_d[l], t[secs], rows*cols);
      }

      //A = A - theta(i)*identity, n_0 = alpha(i)*n_0
      for(i=0; i<0.5*k; i++){
        for(l=0; l<ngpus; l++){
          gpuErrchk(cudaSetDevice(l));
          thetaOp<<<32, 256, 0, streams[l]>>>(newA_d[l], At_d[l],
              theta[i].x, theta[i].y, identity_d[l], rows*cols);
          alphaOp<<<32, 256, 0, streams[l]>>>(newn_0_d[l], alpha[↩
             i].x, alpha[i].y, n_0_d[l], rows);
        }

        //Solve Ax = b, i.e. A\n_0
        #pragma omp parallel private(l)
        {
          l = omp_get_thread_num();
          gpuErrchk(cudaSetDevice(l));
          magma_zgesv_gpu(Mn, Mnrhs, newA_d[l], Mlda, Mpiv[l], ↩
             newn_0_d[l], Mldb, &Minfo[l]);
        }

        // n = n + A\n_0
        for(l=0; l<ngpus; l++){
          gpuErrchk(cudaSetDevice(l));
          Add<<<32, 256, 0, streams[l]>>>(nd_d[l], newn_0_d[l], ↩
             rows);
        }
      }

      //n = 2*real(n), n = n + alpha_0*n_0
      for(l=0; l<ngpus; l++){
        gpuErrchk(cudaSetDevice(l));
        Double<<<32,256, 0, streams[l]>>>(n_d[l], nd_d[l], rows);
        getN<<<32, 256, 0, streams[l]>>>(n_d[l], alpha_0, n_0_d[l↩
           ], rows);

        gpuErrchk(cudaMemcpyAsync(n + secs*rows + l*rows*↩
           timesteps + nitts*rows*timesteps, n_d[l], sizeof(↩
           double)*rows, cudaMemcpyDeviceToHost, streams[l]));
      }
    }
  }

}
```

Appendix C

Custom Maxeler code for matrix multiply

C.1 Host code

```
int main{

  //Allocate memory and initialise matrices

  //Initialise the maxfile
  max_file_t *maxfile = MatMul_init();
  max_engine_t *eng[numEngines];

  //Load each DFE with the maxfile
  for(int i=0; i<numEngines; i++){
    eng[i] = max_load(maxfile, "*");
  }

  //get burst size of LMem
  int burstLengthInBytes = max_get_burst_size(maxfile, "↩
      cmd_tolmem");

  //Structure specifying write LMem parameters, where to start, ↩
      the size of memory to write and the data to write
  MatMul_writeLMem_actions_t *writeAct[nChunks];
```

```
for(int i=0; i<nChunks; i++){
  writeAct[i] = malloc(sizeof(MatMul_writeLMem_actions_t));
  writeAct[i]->param_size = size;
  writeAct[i]->param_start = 0;
  writeAct[i]->instream_fromcpu = B[i];
}

//Structure specifying memory location in LMem to start reading from, the size of memory chunk to read and where to place the read data for one DFE
MatMul_readLMem_actions_t *readAct[nChunks];
for(int i=0; i<nChunks; i++){
    readAct[i] = malloc(sizeof(MatMul_readLMem_actions_t));
    readAct[i]->param_size = sizeR;
    readAct[i]->param_start = size;
    readAct[i]->outstream_tocpu = outData[i];

}

//Structure specifying kernel parameters, scalar input N, K and M, the LMem burstsize and the input stream A, streamed by PCIe
MatMul_actions_t *act[NChunks];
for(int i=0; i<NChunks; i++){
  act[i] = malloc(sizeof(MatMul_actions_t));
  act[i]->param_K = K;
  act[i]->param_M = M;
  act[i]->param_N = N;
  act[i]->param_burstSize = burstLengthInBytes;
  act[i]->instream_A = A[i];
}

//Structure specifying memory location in LMem to start reading from, the size of memory chunk to read and where to place the read data for eight DFEs
MatMul_readLMem_actions_t *readActN[numEngines];
for(int i=0; i<numEngines; i++){
    readActN[i] = malloc(sizeof(MatMul_readLMem_actions_t));
    readActN[i]->param_size = sizeR;
    readActN[i]->param_start = size;
    readActN[i]->outstream_tocpu = outDataN[i];

}

max_run_t *run[numEngines];    //For running kernel
max_run_t *writerun[numEngines];  //For writing LMem
max_run_t *readrun[numEngines];   //For reading LMem

//Running on one DFE
for(int j=0; j<nChunks; j++){
  MatMul_writeLMem_run(eng[0], writeAct[j]);  //Load one chunk of B to LMem
  for(int i=0; i<NChunks; i++){
    MatMul_run(eng[0], act[i]);   //Compute on that chunk with all A chunks
    MatMul_readLMem_run(eng[0], readAct[j]);  //Read each result from LMem
    memcpy(C + i*sizeR + j*sizeRFull, outData[j], sizeBytesR); //Place into final result matrix
  }
}

//Running on 8 DFEs
for(int j=0; j<nChunks; j++){
  //Load each DFE LMem with a different B chunk
  for(int k=0; k<numEngines; k++){
    writerun[k] = MatMul_writeLMem_run_nonblock(eng[k], writeAct[j]);
  }
  //Run the kernel to do the multiplies on each chunk and read result from LMem
  for(int i=0; i<NChunks/numEngines; i++){
    for(int k=0; k<numEngines; k++){
      run[k] = MatMul_run_nonblock(eng[k], act[i*numEngines + k
        ]);
        readrun[k] = MatMul_readLMem_run_nonblock(eng[k], readActN[k]);
      }
      //Wait for reads to finish before moving on to next chunks which will fill the same LMem
      for(int k=0; k<numEngines; k++){
        max_wait(readrun[k]);
      }
      //Put read data into final result matrix
      for(int k=0; k<numEngines; k++){
        memcpy(C8 + k*sizeR + i*sizeR*numEngines + j*FullN*M, outDataN[k], sizeBytesR);
      }
    }
  }

  //Check results...
}
```

C.2 Computational kernel

```
package matmul;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;

class MatMulKernel extends Kernel {

  MatMulKernel(KernelParameters parameters) {
    super(parameters);

    //Define vector of floats with length LMem burstSize
    DFEVectorType<DFEVar> vectorType =
      new DFEVectorType<DFEVar>(dfeFloat(8, 24), 96);

    //Read scalar inputs from CPU specifying size of problem
    DFEVar rowsInA = io.scalarInput("rowsInA",dfeUInt(32)); // N
    DFEVar colsInA = io.scalarInput("colsInA",dfeUInt(32)); // K
    DFEVar colsInBursts = io.scalarInput("colsInBursts",dfeUInt↩
       (32)); // M/burstSize

    //Control counter to read input from A at start of new row of↩
        B
    DFEVar count = control.count.simpleCounter(32, colsInBursts);
    DFEVar readingInput = count === 0;

    //Read input A and B, A controlled by above counter
    DFEVar A = io.input("A", dfeFloat(8, 24), readingInput);
    DFEVector<DFEVar> B = io.input("B", vectorType);

    //Counter chain, equivalent to nested loop with outer loop N,↩
        middle loop K and inner loop M (M/burstSize)
    CounterChain chain = control.count.makeCounterChain();
    DFEVar Arows = chain.addCounter(rowsInA, 1);
    DFEVar Acols = chain.addCounter(colsInA, 1);
    DFEVar Bcols = chain.addCounter(colsInBursts, 1);

    //At the head of the loop, for the first colsInBursts clock ↩
       ticks, set sum = 0.0, otherwise set sum equal to the ↩
       carriedSum being carried around the hardware loop
    DFEVector<DFEVar> carriedSum = vectorType.newInstance(this);
    DFEVector<DFEVar> sum = Acols === 0 ? constant.vect(96, 0.0) ↩
       : carriedSum;

    //Matrix multiplication
    DFEVector<DFEVar> newSum = A*B + sum;

    //Backward edge to connect the new sum back to the head of ↩
       the loop
    carriedSum <== stream.offsetStriped(newSum, -1440);

    //Output result data only when the full column has been ↩
       multiplied and summed
    io.output("oData", newSum, vectorType, Acols===colsInA-1);

  }
}
```

C.3 Manager code

```
package matmul;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.↩
   KernelBlock;
import com.maxeler.maxcompiler.v2.managers.custom.stdlib.↩
   DebugLevel;
import com.maxeler.maxcompiler.v2.managers.custom.stdlib.↩
   MemoryControlGroup;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.↩
   CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.↩
   EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.↩
   EngineInterface.Direction;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.↩
   InterfaceParam;

class MatMulManager extends CustomManager {
```

```

private static final String KERNEL_NAME = "MatMulKernel";
private static final String ControlKERNEL_NAME = "↩
   ControlMatMulKernel";

private static final CPUTypes Type = CPUTypes.FLOAT;
private static final CPUTypes Type1 = CPUTypes.INT32;

MatMulManager(EngineParameters engineParameters) {
  super(engineParameters);

  //Instantiate kernels
  KernelBlock k1 = addKernel(new MatMulKernel(↩
     makeKernelParameters(KERNEL_NAME)));
  KernelBlock k2 = addKernel(new ControlMatMulKernel(↩
     makeKernelParameters(ControlKERNEL_NAME)));
  debug.setDebugLevel(new DebugLevel(){{setHasStreamStatus(true↩
     );}});

  //Instantiate streams to/from CPU and LMem and specify LMem ↩
     memory access pattern
  DFELink fromcpu = addStreamFromCPU("fromcpu");
  DFELink A = addStreamFromCPU("A");
  DFELink tocpu = addStreamToCPU("tocpu");

  DFELink tolmem = addStreamToOnCardMemory("tolmem",↩
     MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
  DFELink fromlmem = addStreamFromOnCardMemory("fromlmem",↩
     MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);

  //Connect the stream from CPU to LMem for writing to LMem and↩
      vice versa for reading from LMem
  tolmem <== fromcpu;
  tocpu <== fromlmem;

  //Set kernel inputs
  DFELink B = addStreamFromOnCardMemory("B",k2.getOutput("↩
      AcmdStream"));
  k1.getInput("A") <== A;
  k1.getInput("B") <== B;

  //Set kernel output
  DFELink oData = addStreamToOnCardMemory("oData",k2.getOutput("OcmdStream"));
  oData <== k1.getOutput("oData");
}

//writes the data from the CPU to the LMem, specifying where to start writing, how many bytes to read and which stream to connect to
private static EngineInterface interfaceWrite(String name) {
  EngineInterface ei = new EngineInterface(name);

  InterfaceParam size  = ei.addParam("size", Type1);
  InterfaceParam start = ei.addParam("start", Type1);
  InterfaceParam sizeInBytes = size * Type.sizeInBytes();

  ei.setStream("fromcpu", Type, sizeInBytes );
  ei.setLMemLinear("tolmem", start * Type.sizeInBytes(), sizeInBytes);
  ei.ignoreAll(Direction.IN_OUT);
  return ei;
}

//reads the data back to the CPU from the LMem, specifying where to start reading, how many bytes to read and which stream to connect to
private static EngineInterface interfaceRead(String name) {
  EngineInterface ei = new EngineInterface(name);

  InterfaceParam size  = ei.addParam("size", Type1);
  InterfaceParam start = ei.addParam("start", Type1);
  InterfaceParam sizeInBytes = size * Type.sizeInBytes();

  ei.setLMemLinear("fromlmem", start * Type.sizeInBytes(), ↩
     sizeInBytes);
  ei.setStream("tocpu", Type, sizeInBytes);
  ei.ignoreAll(Direction.IN_OUT);
  return ei;
}

//Sets the kernel parameters, the scalar inputs, number of ↩
   clock ticks for the kernel to run and the PCIe stream A
private static EngineInterface interfaceDefault() {
  EngineInterface ei = new EngineInterface();

  InterfaceParam N   = ei.addParam("N", Type1);
  InterfaceParam K    = ei.addParam("K", Type1);
  InterfaceParam M    = ei.addParam("M", Type1);
  InterfaceParam burstSize = ei.addParam("burstSize", Type1);
  ei.setTicks(KERNEL_NAME, (N*K*M*(Type.sizeInBytes()))/↩
     burstSize);
  ei.setTicks(ContolKERNEL_NAME, (N*K*M*(Type.sizeInBytes()))/↩
     burstSize);
  ei.setStream("A", Type, N*K*Type.sizeInBytes());

  ei.setScalar(ControlKERNEL_NAME, "totalBursts", (K*M*(Type.↩
     sizeInBytes()))/burstSize);
  ei.setScalar(ContolKERNEL_NAME, "rowsInA", N);
  ei.setScalar(ControlKERNEL_NAME, "colsInBursts", (M*(Type.↩
     sizeInBytes()))/burstSize);
  ei.setScalar(ControlKERNEL_NAME, "colsInA", K);
  ei.setScalar(KERNEL_NAME, "rowsInA", N);
  ei.setScalar(KERNEL_NAME, "colsInA", K);
  ei.setScalar(KERNEL_NAME, "colsInBursts", (M*(Type.↩
     sizeInBytes()))/burstSize);

  ei.setLMemInterruptOn("oData");
  ei.ignoreAll(Direction.IN_OUT);
  return ei;
}

//Creates SLiC interfaces to provide DFE functions to be called from CPU
public static void main(String[] args) {
  MatMulManager m = new MatMulManager(new EngineParameters(args));
  m.createSLiCinterface(interfaceRead("readLMem"));
  m.createSLiCinterface(interfaceWrite("writeLMem"));
  m.createSLiCinterface(interfaceDefault());

  m.build();
}
}
```

Bibliography

[1] G.E. MOORE, “Cramming more components onto integrated circuits”, *Electronics Magazine*, **38**, 114 (1965)

[2] A. BÉCOULET et al., “The way towards thermonuclear fusion simulators”, *Computer Physics Communications*, **177**, 55 (2007)

[3] A. LENARD and I.B. BERNSTEIN, “Plasma oscillations with diffusion in velocity space”, *Physical Review*, **112**, 1456 (1958)

[4] J.M. DAWSON, “Computer modelling of plasma: Past, present, and future”, *Physics of Plasmas*, **2**, 2189 (1995)

[5] J.M. DAWSON, “Onedimensional plasma model”, *Physics of Fluids*, **5**, 445 (1962)

[6] O. BUNEMAN, “Dissipation of currents in ionized media”, *Physical Review*, **115**, 503 (1959)

[7] R.W. HOCKNEY, “A fast direct solution of Poisson’s equation using Fourier analysis”, *Journal of the Association for Computing Machinery*, **12**, 95 (1965)

[8] R.W. HOCKNEY, “Computer experiment of anomalous diffusion”, *The Physics of Fluids*, **9**, 1826 (1966)

[9] O. BUNEMAN, “The advance from 2D electrostatic to 3D electromagnetic particle simulation”, *Computer Physics Communications*, **12**, 21 (1976)

[10] J.M. DAWSON, “Contribution of computer simulation to plasma theory”, *Computer Physics Communications*, **3**, 79 (1972)

[11] R. BUNEMAN et al., “A tribute to Oscar Buneman-Pioneer of plasma simulation”, *IEEE Transactions on Plasma Science*, **22** 22 (1994)

[12] O. BUNEMAN et al., “Principles and capabilites of 3-D, E-M particle simulations”, *Journal of Computational Physics*, **38**, 1 (1980)

[13] B.I. COHEN et al., “The numerical tokamak project: simulation of turbulent transport”, *Computer Physics Communications*, **87**, 1 (1995)

[14] J.M. DAWSON et al., “Physics modelling of Tokamak transport, a grand challenge for controlled fusion”, *The International Journal of Supercomputer Applications*, **5**, 13 (1991)

[15] J.M. DAWSON, “Role of computer modeling of plasmas in the 21st century”, *Physics of Plasmas*, **6**, 4436 (1999)

[16] R.D. SYDORA et al., “Fluctuation-induced heat transport results from a large global 3D toroidal particle simulation model”, *Plamsa Physics and Controlled Fusion*, **38**, A281 (1996)

[17] J. WANG et al., “3D electromagnetic plasma particle simulations on a MIMD parallel computer”, *Computer Physics Communications*, **87**, 35 (1995)

[18] J.N. LEBOEUF et al., “Effect of externally imposed and self-generated flows on turbulence and magnetohydrodynamic activity in tokamak plasmas”, *Physics of Plasmas*, **7**, 1795 (2000)

[19] J.B. TAYLOR and B. MCNAMARA, “Plasma diffusion in two dimensions”, *Physics of Fluids*, **14**, 1492 (1971)

[20] W. PARK et al., “Threedimensional hybrid gyrokineticmagnetohydrodynamics simulation”, *Physics of Fluids B*, **4** 2033 (1992)

[21] Z. LIN et al., “Turbulent transport reduction by zonal flows: massively parallel simulations”, *Science*, **281**, 1835 (1998)

[22] M. KOTSCHENREUTHER et al., "Comparison of initial value and eigenvalue codes for kinetic toroidal plasma instabilities", *Computer Physics Communications*, **88**, 128 (1995)

[23] W. DORLAND et al., "Electron temperature gradient turbulence", *Physical Review Letters*, **85**, 5579 (2000)

[24] T. GÖRLER et al., "The global version of the gyrokinetic turbulence code GENE", *Journal of Computational Physics*, **230**, 7053 (2011)

[25] J. CANDY and R.E. WALTZ, "An Eulerian gyrokinetic-Maxwell solver", *Journal of Computational Physics*, **186**, 545 (2003)

[26] S. ETHIER et al., "Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms", *Journal of Physics: Conference Series*, **16**, 1 (2005)

[27] M. BARNES, "Trinity: A unified treatment of turbulence, transport, and heating in magnetized plasmas", *PhD thesis*, University of Maryland, Maryland, (2008)

[28] M. BARNES, "Direct multiscale coupling of a transport code to gyrokinetic turbulence codes", *Physics of Plasmas*, **17**, 056109 (2010)

[29] P. WORLEY et al., "Performance analysis of GYRO: a tool evaluation", *Journal of Physics: Conference Series*, **16**, 551 (2005)

[30] A. JACKSON et al., "Optimising performance through unbalanced decompositions", *IEEE Transactions on Parallel and Distributed Systems*, **26**, 2863 (2015)

[31] X.Q. XU, "The BOUT project; validation and benchmark of BOUT code and experimental diagnostic tools for fusion boundary turbulence", *Plasma Science and Technology*, **3**, 959 (2001)

[32] H.R. WILSON et al., "Numerical studies of edge localized instabilities in tokamaks", *Physics of Plasmas*, **9**, 1277 (2002)

[33] P.B. SNYDER et al., "Edge localized modes and the pedestal: A model based on coupled peelingballooning modes", *Physics of Plasmas*, **9**, 2037 (2002)

[34] G.T.A. HUYSMANS and O. CZARNY, "MHD stability in X-point geometry: simulation of ELMs", *Nuclear Fusion*, **47**, 659 (2007)

[35] B.D. DUDSON et al., "BOUT++: A framework for parallel plasma fluid simulations", *Computer Physics Communications*, **180**, 1467 (2009)

[36] O. MOTOJIMA, "The ITER project construction status", *Nuclear Fusion*, **55**, 104023 (2015)

[37] A.W. MORRIS et al., "Spherical tokamaks: Present status and role in the development of fusion power", *Fusion Engineering and Design*, **74**, 67 (2005)

[38] B.D. DUDSON et al., "BOUT++: Recent and current developments", *Journal of Plasma Physics*, **81**, 365810104 (2015)

[39] K.Z. IBRAHIM et al., "Analysis and optimization of gyrokinetic toroidal simulations on homogenous and heterogenous platforms", *International Journal of High Performance Computing Applications*, **27**, 454 (2013)

[40] T. DANNERT et al., "Porting large HPC applications to GPU clusters: The codes GENE and VERTEX", *Parallel Computing: Accelerating Computational Science and Engineering*, IOS Press, 305 (2014)

[41] K. GERMASCHEWSKI et al., "The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing", *Journal of Computational Physics*, **318**, 305 (2016)

[42] J.M. DAWSON, "Particle simulation of plasmas", *Reviews of Modern Physics*, **55**, 403 (1983)

[43] J.P. VERBONCOEUR, "Particle simulation of plasmas: review and advances", *Plasma Physics and Controlled Fusion*, **47**, A231 (2005)

[44] F. HARIRI et al., “A portable platform for accelerated PIC codes and its application to GPUs using OpenACC”, *Computer Physics Communications*, **207**, 69 (2016)

[45] J.D. OWENS et al., “A survey of general-purpose computation on graphics hardware”, *Computer Graphics Forum*, **26**, 80 (2007)

[46] R. LI and Y. SAAD, “GPU-accelerated preconditioned iterative linear solvers”, *The Journal of Supercomputing*, **63**, 443 (2013)

[47] G.R. MORRIS and K.H. ABED, “Mapping a Jacobi iterative solver onto a high-performance heterogeneous computer”, *IEEE Transactions on Parallel and Distributed Systems*, **24**, 85 (2013)

[48] V.K. DECYK and T.V. SINGH, “Adaptable Particle-in-Cell algorithms for graphical processing units”, *Computer Physics Communications*, **182**, 641 (2011)

[49] H. GIEFERS et al., “Accelerating finite difference time domain simulations with reconfigurable dataflow computers”, *Fourth International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, (2013)

[50] L. YANG et al., “High Performance Data Clustering: A Comparative Analysis of Performance for GPU, RASC, MPI and OpenMP Implementations”, *The Journal of Supercomputing*, **70**, 284 (2014)

[51] K. THOUTI and S.R. SATHE, “Comparison of OpenMp and OpenCL Parallel Processing Technologies”, *International Journal of Advanced Computer Science and Applications*, **3**, 56 (2012)

[52] E. MONTEIRO et al., “Parallelization of Full Search Motion Estimation Algorithm for Parallel and Distributed Platforms”, *International Journal of Parallel Programming*, **42**, 239 (2014)

[53] J. DONGARRA et al., “The international exascale software project roadmap”, *The International Journal of High Performance Computing Applicaitons*, **25**, 3 (2011)

[54] C.A. NAVARRO et al., "A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures", *Communications in Computational Physics*, **15**, 285 (2014)

[55] A. SCLOCCO et al., "Radio Astronomy Beam Forming on Many-Core Architectures", *IEEE 26th International Parallel and Distributed Processing Symposium*, **26**, 1105 (2012)

[56] E. VERMIJ et al., "Challenges in exascale radio astronomy: can the SKA ride the technology wave?", *The International Journal of High Performance Computing Applications*, **29**, 37 (2015)

[57] P. LUJAN et al., "GPU Enhancement of the Trigger to Extend Physics Reach at the LHC", *Journal of Physics: Conference Series*, **513**, 012019 (2014)

[58] P. THIAGARAJ et al., "FPGA processing for High Performance Computing", *Emerging Technology Conference*, Manchester, 30th June - 1st July (2015)

[59] N. GARELLI, "The Evolution of the Trigger and Data Acquisition System in the ATLAS Experiment", *Journal of Physics: Conference Series*, **513**, 012007 (2014)

[60] O. PELL and V. AVERBUKH, "Maximum Performance Computing with Dataflow Engines", *Computing in Science and Engineering*, **14**, 98 (2012)

[61] J.D. CAPPELLO and D. STRENSKI, "A Practical Measure of FPGA Floating Point Acceleration for High Performance Computing", *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, **24**, 160 (2013)

[62] "Top500 Lists June 2015", www.top500.org/lists/2015/06 (June 2015)

[63] C. XU et al., "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the Tianhe-1A supercomputer", *Journal of Computational Physics*, **278**, 275, (2014)

[64] J. NIETO et al., "A GPU-based real time high performance computing service in a fast plant system controller prototype for ITER", *Fusion Engineering and Design*, **87**, 2152 (2012)

[65] W.M. DAVIS et al., "Storage and analysis techniques for fast 2-D camera data on NSTX", *Fusion Engineering and Design*, **81**, 1975 (2006)

[66] J.B. LISTER et al., "The ITER project and its Data Handling Requirements", *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, **9**, 589 (2003)

[67] R. CASTRO et al., "Soft real-time EPICS extensions for fast control: A case study applied to a TCV equilibrium algorithm", *Fusion Engineering and Design*, **89**, 638 (2014)

[68] R.J. AKERS et al., "GPGPU Monte Carlo calculation of gyro-phase resolved fast ion and n-state resolved neutral deuterium distributions", *39th EPS Conference and 16th International Congress on Plasma Physics*, P5.088 (2012)

[69] A.G. CHIARIELLO et al., "Fast magnetic field computation in fusion technology using GPU technology", *Fusion Engineering and Design*, **88**, 1635 (2013)

[70] X.YUE et al., "Fast equilibrium reconstruction for tokamak discharge control based on GPU", *Plasma Physics and Controlled Fusion*, **55**, 085016 (2013)

[71] N. RATH et al., "High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak", *Fusion Engineering and Design*, **87**, 1895 (2012)

[72] J. NIETO et al., "Exploiting graphic processing units parallelism to improve intelligent data acquisition system performance in JETs correlation reflectometer", *17th IEEE NPSS Real Time Conference*, Lisbon (2010)

[73] H. NAKANISHI et al., "Portability improvement of LABCOM data acquisition system for the next-generation fusion experiments", *Fusion Engineering and Design*, **82**, 1203 (2007)

[74] L. ESTEBAN et al., “Continuous plasma density measurement in TJ-II infrared interferometerAdvanced signal processing based on FPGAs”, *Fusion Engineering and Design*, **85**, 328 (2010)

[75] M. RUIZ et al., “Real time plasma disruptions detection in JET implemented with the ITMS platform using FPGA based IDAQ”, *IEEE Transactions on Nuclear Science*, **58**, 1576 (2011)

[76] G.A. RATTÁ et al., “An advanced disruption predictor for JET tested in a simulated real-time environment”, *Nuclear Fusion*, **50** (2010)

[77] G.A. NAYLOR, “An FPGA based control unit for synchronization of laser Thomson scattering measurements to plasma events on MAST”, *Fusion Engineering and Design*, **85**, 280 (2010)

[78] R.C. PEREIRA et al., “Enhanced neutron diagnostics data acquisition system based on a time digitizer and transient recorder hybrid module”, *Fusion Engineering and Design*, **81**, 1873 (2006)

[79] V. MARTIN et al., “New field programmable gate array-based image-oriented acquisition and real-time processing applied to plasma facing component thermal monitoring”, *Review of Scientific Instruments*, **81**, 10E113 (2010)

[80] J.C. CHORLEY et al., “GPU-based data processing for 2-D microwave imaging on MAST”, *Fusion Science and Technology*, **69**, 643 (2016)

[81] H. ZOHM et al., “On the physics guidelines for a tokamak DEMO”, *Nuclear Fusion*, **53**, 073019 (2013)

[82] L. DAGUM and R. MENON, “OpenMP: an industry standard API for shared-memory programming”, *Computational Science and Engineering*, **5**, 46 (1998)

[83] M.J. QUINN, “Parallel Programming in C with MPI and OpenMP”, *McGraw-Hill* (2003)

[84] J.D. OWENS et al., “GPU Computing”, *Proceedings of the IEEE*, **96**, 879 (2008)

[85] NAG, "Accelerating Applications with CUDA", *CCFE Workshop*, Nov (2013)

[86] E. WYNTERS, "Parallel processing on Nvidia graphics processing units using CUDA", *Journal of Computing Sciences in Colleges*, **26**, 58 (2011)

[87] J. NICKOLLS and W.J. DALLY, "The GPU Computing Era", *IEEE Computer Society*, **30**, 56 (2010)

[88] Y. ELKURDI et al., "FPGA architecture and implementation of sparse matrixvector multiplication for the finite element method", *Computer Physics Communications*, **178** 558 (2008) (2008)

[89] F. PRATAS et al., "Accelerating the computation of induced dipoles for molecular mechanics with dataflow engines", *21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines* (2013)

[90] V. RANKOVIC et al., "Performance of the bitonic mergesort network on a dataflow computer", *21st Telecommunications Forum TELFOR* (2013)

[91] X. NIU et al., "A scalable design approach for stencil computation on reconfigurable clusters", *23rd International Conference on Field Programmable Logic and Applications* (2013)

[92] O. PELL et al., "Finite-difference wave propagation modelling on special purpose dataflow machines", *IEEE Transactions on Parallel and Distributed Systems* **24** (2013)

[93] The Green500: http://www.green500.org

[94] Maxeler Technologies, "Multiscale dataflow computing", *Multiscale Dataflow Programming*, **2014.1a** (2014)

[95] Maxeler Technologies, "Maxeler acceleration technology", *MaxCompiler White Paper* (2011)

[96] R. CATTANEO et al., "Runtime adaption on dataflow HPC platforms", *NASA/ESA Conference on Adaptive Hardware and Systems* (2013)

[97] S. STOJANOVIC et al., “An overview of selected hybrid and reconfigurable architectures”, *IEEE International Conference on Industrial Technology* (2012)

[98] Maxeler Technologies, “Accelerating 3D finite difference”, *MaxGenFD White Paper* (2012)

[99] A. VAPIREV et al., “Initial results on computational performance of Intel many integrated core, sandy bridge, and graphical processing unit architectures: implementation of a 1D C++/OpenMP electrostatic particle-in-cell code”, *Concurrency and Computation: Practice and Experience*, **27**, 581 (2015)

[100] C. CULLINAN et al., “Computing Performance Benchmarks among CPU, GPU and FPGA”, (2012)

[101] T. CRAMER et al., “OpenMP programming on Intel Xeon Phi Coprocessors: An early performance comparison”, *Proceeedings of the Many-core Applications Research Community Symposium*, (2012)

[102] S. KESTUR et al., “Blas comparison on FPGA, CPU and GPU”, *Proceedings of IEEE Computer Society Annual Symposium*, 288 (2010)

[103] J. FOWERS et al., “A performance and energy comparison of FPGAs, GPUs and multicores for sliding window applications”, *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 47 (2012)

[104] V.F. SHEVCHENKO et al., “Synthetic aperture microwave imaging with active probing for fusion plasma diagnostics”, *Journal of Instrumentation*, **7**, P10016 (2012)

[105] I.B. BERNSTEIN, “Waves in a plasma in a magnetic field”, *Physical Review*, **109**, 10 (1958)

[106] J. PREINHAELTER et al., “Electron Bernstein wave-X-O mode conversion and electron cyclotron emission in MAST”, *Review of Scientific Instruments*, **74**, 1437 (2003)

[107] S.J. FREETHY et al., “Lensless passive and active microwave imaging on MAST”, *Plasma Physics and Controlled Fusion*, **55**, 124010 (2013)

[108] M.A. ASGARIAN et al., “Direct X-B mode conversion for high-β national spherical torus experiment in non-linear regime”, *Physics of Plasmas*, **21**, 092516 (2014)

[109] M.A. ASGARIAN et al., “Kinetic simulation of the O-X conversion process in dense magnetized plasmas”, *Physics of Plasmas*, **20**, 102516 (2013)

[110] H. IGAMI et al., “Electron Bernstein wave heating via the slow X-B mode conversion process with direct launching from the high field side in LHD”, *Nuclear Fusion*, **49**, 115005 (2009)

[111] A. KÖHN et al., “Visualization of the O-X-B mode conversion process with a full wave code”, *IEEE Transactions on Plasma Science*, **36**, 1220 (2008)

[112] A.D. PILIYA and A.Y. POPOV, “On application of the reciprocity theorem to calculation of a microwave radiation signal in inhomogenous hot magnetized plasmas”, *Plasma Physics and Controlled Fusion*, **44**, 467 (2002)

[113] P.H. VAN CITTERT, “Die Wahrscheinliche Schwingungsverteilung in Einer von Einer Lichtquelle Direkt Oder Mittels Einer Linse Beleuchteten Ebene”, *Physica*, **1**, 201 (1934)

[114] F. ZERNIKE, “The concept of degree of coherence and its application to optical problems”, *Physica*, **5**, 785 (1938)

[115] S.J. FREETHY, “Synthetic aperture imaging of B-X-O mode conversion”, *PhD Thesis*, (2012)

[116] B.K. HUANG et al., “FPGA-based embedded Linux technology in fusion: The MAST microwave imaging system”, *Fusion Engineering and Design*, **87**, 2106 (2012)

[117] S.J. FREETHY et al., “Optimization of wide field interferometric arrays via simulated annealing of a beam efficiency function”, *IEEE transactions on antennas and propagation*, **60**, 5442 (2012)

[118] D.A. THOMAS et al., “2D Doppler backscattering using synthetic aperture microwave imaging of MAST edge plasmas”, *Nuclear Fusion*, **56** 026013 (2016)

[119] M. ONO et al., “Progress toward commissioning and plasma operation in NSTX-U”, *Nuclear Fusion*, **55**, 073007 (2015)

[120] W. MORRIS et al., “MAST accomplishments and upgrade for fusion next-steps”, *IEEE Transactions on Plasma Science*, **42**, 402 (2014)

[121] “ITER Physics Basis”, *Nuclear Fusion*, **39**, 2175 (1999)

[122] N.W. EIDIETIS et al., “The ITPA disruption database”, *Nuclear Fusion*, **55**, 063030 (2015)

[123] G. CUNNINGHAM, “Use of the absolute phase in frequency modulated continuous wave plasma reflectometry”, *Review of Scientific Instruments*, **79**, 083501 (2008)

[124] S.J. DIEM et al., “Investigation of electron Bernstein wave (EBW) coupling and its critical dependence on EBW collisional loss in high- , H-mode ST plasmas”, *Nuclear Fusion*, **49**, 095027 (2009)

[125] S.J. DIEM et al., “Collisional damping of electron Bernstein waves and its mitigation by evaporated lithium conditioning in spherical-tokamak plasmas”, *Physical Review Letters*, **103**, 015002 (2009)

[126] S. SHIBAEV et al., “Real time operation of MAST Thomson scattering diagnostic”, *IEEE Transactions on Nuclear Science*, **58**, 1516 (2011)

[127] S.J. ZWEBEN et al., “Edge turbulence imaging on NSTX and Alcator C-Mod”, *29th EPS Conference on Plasma Physics and Controlled Fusion*, **26B**, June 2002

[128] K.J. GIBSON et al., “New physics capabilities from the upgraded Thomson scattering diagnostic on MAST”, *Plasma Physics and Controlled Fusion*, **52**, 124041 (2010)

[129] E. MJØLHUS, “Coupling to Z mode near critical angle”, *Journal of Plasma Physics*, **31**, 7 (1984)

[130] H.P. LAQUA, “Electron Bernstein wave heating and diagnostic”, *Plasma Physics and Controlled Fusion*, **49**, R1 (2007)

[131] J. WESSON, “Tokamaks”, *International Series of Monographs on Physics*, **149**, (2011)

[132] H. BATEMAN, “Solution of a system of differential equations occurring in the theory of radio-active transformations”, *Proc. Cambridge Phil.*, **423** (1910)

[133] J.C.C. SUBLET et al., “The FISPACT-II User Manual”, *CCFE-R(11)11*, **6** (2014)

[134] L. MORGAN, “Inertial confinement fusion neutronics”, *PhD Thesis*, (2012)

[135] L. MURRAY, “GPU acceleration of Runge-Kutta integrators”, *IEEE Transactions on Parallel and Distributed Systems*, **23**, 94 (2012)

[136] A. HIEMLICH et al., “Parallel GPU implementation of PWR reactor burnup”, *Annals of Nuclear Energy*, **91**, 135 (2016)

[137] C. MOLER and C. VAN LOAN, “Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later,” *SIAM Rev.*, **45** (2003)

[138] D. TAYLOR et al., “Neutron shielding and activation of the MASTU device and surrounds”, *Fusion Engineering and Design*, **89**, 2076 (2014)

[139] J. SHIMWELL et al., “Spatially and temporally varying tritium generation in solid-type breeder blankets”, *Fusion Engineering and Design*, **98-99**, 1868 (2015)

[140] A.J. KONING et al., "TENDL-2015: TALYS-based evaluated nuclear data library",
https://tendl.web.psi.ch/tendl_2015/tendl2015.html, (2016)

[141] A.J. KONING and D. ROCHMAN, "Modern nuclear data evaluation with the TALYS code system", *Nuclear Data Sheets*, **113**, 2841 (2012)

[142] Cross Sections Evaluation Working Group, "ENDF-6 Formats Manual", *Brookhaven National Laboratory*, July (2010)

[143] K. RADHAKRISHNAN and A.C. HINDMARSH, "Description and use of LSODE, the Livermore solver for ordinary differential equations", *LLNL Report UCRL-ID-113855*, LLNL, (1993)

[144] C.W. GEAR, "Numerical initial value problems in ordinary differential equations", *Prentice Hall*, New Jersey (1971)

[145] F. BASHFORTH and J.C. ADAMS, "An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid, with an explanation of the method of integration employed in constructing the tables which give the theoretical forms of such drops", *Cambridge University Press*, Cambridge, (1883)

[146] F.R. MOULTON, "Discussions: New Methods in Exterior Balistics", *The American Mathematical Monthly*, **35**, 246 (1928)

[147] T.D. BUI et al., "Recent advances in methods for numerical solution of O.D.E. initial value problems", *Journal of Computational and Applied Mathematics*, **11**, 283 (1984)

[148] NVIDIA, "cuBLAS :: CUDA toolkit documentation",
http://docs.nvidia.com/cuda/cublas/#axzz4G4hlGL44, (2016)

[149] EUROfusion, "Inconel 600",
https://www.euro-fusion.org/2012/01/inconel-600/ (2016)

[150] G. FEDERICI et al., “European DEMO design strategy and consequences for materials”, *preprint available at:* http://www.euro-fusionscipub.org/wp-content/uploads/2015/12/WPMATPR1529.pdf

[151] M.R. GILBERT and J.C. SUBLET, “Scoping of material response under DEMO neutron irradiation: comparison with fission and influence of nuclear library selection”, *preprint available at:* https://arxiv.org/pdf/1604.08496.pdf, *to be published in Fusion Engineering and Design*, December (2016)

[152] M. PUSA and J. LEPPÄNEN, “Computing the matrix exponential in burnup calculations”, *Nuclear Science and Engineering*, **164** (2010)

[153] M. PUSA, “Rational approximations to the matrix exponential in burnup calculations”, *Nuclear Science and Engineering*, **169** (2011)

[154] W.J. CODY et al., “Chebyshev rational approximations to e^{-x} in $[0, +\infty)$ and applications to heat-conduction problems”, *Journal of Approximation Theory*, **2**, 50 (1969)

[155] MATLAB - MathWorks, “Documentation - mldivide, \”, http://uk.mathworks .com/help/matlab/ref/mldivide.html (2016)

[156] T.A. DAVIS, “Algorithm 832: UMFPACK V4.3An unsymmetric-pattern multifrontal method”, *ACM Transactions on Mathematical Software*, **30**, 196 (2004)

[157] Y. CHEN et al., “Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate”, *ACM Transactions on Mathematical Software*, **35**, 22 (2008)

[158] T.A. DAVIS and W.W. HAGER, “Dynamic supernodes in sparse Cholesky update/downdate and triangular solves”, *ACM Transactions on Mathematical Software*, **35**, 27 (2009)

[159] T.A. DAVIS and E.P. NATARAJAN, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems", *ACM Transactions on Mathematical Software*, **37**, 36 (2010)

[160] MATLAB - MathWorks, "Matrix exponential - expm", http://uk.mathworks.com/help/matlab/ref/expm.html (2016)

[161] MATLAB - MathWorks, "Matrix exponentials - expmdemo1, expmdemo2 and expmdemo3", http://www.mathworks.com/examples/matlab/mw/matlab_featured-ex73265356-matrix-exponentials (2016)

[162] E. AGULLO et al., "Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects", *Journal of Physics: Conference Series*, **180**, 012037 (2009)

[163] MAGMA 1.7.0, "Matrix Algebra for GPU and Multicore Architectures", http://icl.cs.utk.edu/projectsfiles/magma/doxygen/index.html (2015)

[164] NVIDIA, "cuSPARSE :: CUDA toolkit documentation", http://docs.nvidia.com/cuda/cusparse/#axzz4G4hlGL44 (2016)

[165] MAGMA 1.7.0, "LU solve: driver", http://icl.cs.utk.edu/projectsfiles/magma/doxygen/group__magma__zgesv__driver.html (2015)

[166] S. TOMOV et al., "Dense linear algebra solvers for multicore with GPU accelerators", *IEEE International Symposium on Parallel and Distributed Processing*, 19-23 April (2010)

[167] E. AGULLO et al., "LU factorization for accelerator-based systems", *9th IEEE/ACS International Conference on Computer Systems and Applications*, 27-30 December (2011)

[168] A. HAIDAR et al., "Batched matrix computations on hardware accelerators based on GPUs", *The International Journal of High Performance Computing Applications*, **29**, 193 (2015)

[169] STFC, "Emerald", *Science and Engineering South*, http://www.ses.ac.uk/high-performance-computing/emerald/ (2016)

[170] MAGMA 1.7.0, "The MAGMA sparse-iter package", http://icl.cs.utk.edu/projectsfiles/magma/doxygen/sparse-iter.html (2016)

[171] K. LI et al., "Fast and processor efficient parallel matrix multiplication algorithms on a linear array with a reconfigurable pipelined bus system", *IEEE Transactions on Parallel and Distributed Systems*, **9**, 705 (1998)

[172] A. BRANDT and A.A LUBRECHT, "Multilevel matrix multiplication and fast solution of integral equations", *Journal of Computational Physics*, **90**, 348 (1990)

[173] Y. HASEGAWA et al., "Performance evaluation of ultra large-scale first-principles electronic structure calculation code on the K computer", *The International Journal of High Performance Computing Applications*, **28**, 335 (2014)

[174] A.B. LANGDON et al., "Direct implicit large time-step particle simulation of plasmas", *Journal of Computational Physics*, **51**, 107 (1983)

[175] O. PELL and O. MENCER, "Surviving the end of frequency scaling with reconfigurable dataflow computing", *ACM SIGARCH Computer Architecture News*, **39**, 60 (2011)

[176] Intel Math Kernel Library (Intel MKL): https://software.intel.com/en-us/intel-mkl

[177] X. JIANG and J. TAO, "Implemenation of effective matrix multiplication on FPGA", *Proceedings of IEEE IC-BNMT*, **4**, 656 (2011)

[178] Simulink - MathWorks, " Simulation and model-based design", http://uk.mathworks.com/products/simulink/

[179] T. ZHANG et al., "An optimized floating-point matrix multiplication on FPGA", *Information Technology Journal*, **12**, 1832 (2013)

[180] N. DAVE et al., "Hardware acceleration of matrix multiplication on a Xilinx FPGA", *IEEE/ACM IC-FMMC*, **5**, 97 (2007)

[181] A. KHAYYAT and N. MANJIKIAN, "Analysis of blocking and scheduling for FPGA-based floating-point matrix multiplication", *Canadian Journal of Electrical and Computer Engineering*, **37**, 65 (2014)

[182] J.W. JANG et al., "Energy and time-efficient matrix multiplication on FPGAs", *IEEE Transactions on Very Large Scale Integration Systems*, **13**, 1305 (2005)

[183] The Maxeler App Gallery, http://appgallery.maxeler.com, (2016)

[184] Maxeler Technologies, "STFC Daresbury Laboratory first to install Maximum Performance Computer (MPC)",
https://www.maxeler.com/stfc-dataflow-supercomputer (February 2014)

[185] Science and Technologies Facilities Council, "STFC in new collaboration with aim of creating one of the world's most energy efficient supercomputers",
http://www.stfc.ac.uk/news/stfc-in-new-collaboration-with-aim-of-creating-one-of-the-worlde28099s-most-energy-efficient-supercomputers (February 2014)

[186] Maxeler Technologies, "MPC-X Series",
https://www.maxeler.com/products/mpc-xseries (2016)

[187] V. VOLKOV and J.W. DEMMEL, "Benchmarking GPUs to tune dense linear algebra", *International Conference for High Performance Computing, Networking, Storage and Analysis*, **20**, November (2008)

[188] C. JONES, "Dense Matrix Multiplication", *MaxAppGallery* (2016)

[189] B. SUBRAMANIAM et al., "Trends in energy-efficient computing: A perspective from the Green500", *International Green Computing Conference*, **4**, June (2013)

[190] J.G. KOOMEY et al., “Implications of historical trends in the electrical efficiency of computing”, *IEEE Annals of the History of Computing*, **33**, 46 (2011)

[191] J. CARABAÑO et al., “An exploration of heterogeneous systems”, *International Workshop on Reconfigurable and Communication-Centric Systems-On-Chip*, **8**, (2013)

[192] J. CHEN et al., “FPGA-accelerated 3D reconstruction using compressive sensing”, *Proceedings of the ACM-SIGDA International Symposium on Field Programmable Gate Arrays*, **FPGA12**, 163 (2012)

[193] Olsen Electronics Ltd, “Meter Range”,
http://www.olson.co.uk/meters.htm

[194] B. BETKAOUI et al., “Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing”, *International Conference on Field Programmable Technology*, **FPT10**, 94 (2010)

[195] P. MUCCI et al., “Performance Application Programming Interface - PAPI”, University of Tennessee - Innovative Computing Laboratory, http://icl.cs.utk.edu/papi/, January 2016

[196] A. LÖSCH and C. KNORR, “Accurately Measuring Power and Energy for Heterogeneous Resource Environments - Ampehre”, University of Paderborn, https://github.com/akiml/ampehre, March 2016]

[197] Y. SATO et al., “Evaluating reconfigurable dataflow computing using the Himeno benchmark”, *International Conference on Reconfigurable Computing and FPGAs*, **10**, December (2012)

[198] C. TRINITIS, “BLAS routines implementations on Maxeler platforms”, *OpenSPL Symposium* (2014)

[199] XILINX, “Introduction to FPGA Design with Vivado HLS”, *Xilinx Documentation*, **1.0**, July (2013)

[200] J. CONG et al., "High-level synthesis for FPGAs: From prototyping to deployment", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **30**, 473 (2011)]

[201] C. BRUGGER et al., "A quantitative cross-architecture study of morphological image processing on CPUs, GPUs, and FPGAs", *IEEE Symposium on Computer Applications and Industrial Electronics*, 201 (2015)

[202] S. WINDH et al. "High-level language tools for reconfigurable computing", *Proceedings of the IEEE*, **103**, 390 (2015)

[203] B. WILE, "Coherent Accelerator Processor Interface (CAPI) for POWER8 systems", *IBM - White Paper*, September (2014)

[204] N. HEMSOTH, "Intel marrying FPGA, beefy Broadwell for open compute future", *The Next Platform*, March (2016)

[205] Xilinx, "Zynq-7000 all programmable SoC", https://www.xilinx.com/publications/prod_mktg/zynq-7000-generation-ahead-backgrounder.pdf