Leveraging Many-Core Technology for Deterministic Neutral Particle Transport at Extreme Scale

Tom Deakin

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

January 2018

50,300 words.

Abstract

With disruptive changes to supercomputing architecture at the node level, algorithms are required to leverage the increased parallelism and high bandwidth memory technologies from many-core devices. The deterministic discrete ordinates transport equation is an important equation which models the movement and interaction of neutral particles, such as neutrons, through materials. The balance equation counts the loss and gain of the particles through changes in direction, energy, and as a result of collisions with material nuclei. Solving this equation cannot be performed analytically for all but the simplest problems and so in practice the solution must be approximated using numerical methods.

The Discrete Ordinates (S_n) discretisation used in solving the equation numerically imposes a wavefront dependency across the spatial domain, and as such there is a corresponding limitation on the concurrency of the algorithm. The sweep and finite difference discretisation of the spatial domain result in complex data reuse patterns. The problem is of high dimensionality, with angular, energy and spatial domains modelled over time. Therefore the solution itself has a high memory footprint so that it often becomes bound by the available memory capacity of supercomputer nodes. All these factors mean that exploiting many-core technology is a significant challenge.

This thesis will investigate solving the transport equation on many-core architectures. Performance models will be developed in order to capture the behaviour of the memory accesses and communication patterns. A GPU implementation of a transport mini-app will be developed using a concurrent scheme which demonstrates for the first time that such devices can be used to provide speedups. The reduction in runtime is in line with the memory bandwidth advantages GPUs have over CPU architectures. Mini-apps will be developed to capture the critical computation to examine the solver on cache-based architectures. Finally a high order discontinuous Galerkin finite element method will be investigated in order to mitigate memory capacity constraints of high bandwidth memories at an algorithmic level.

Dedication

My thanks my first and foremost go to my supervisor, Prof Simon McIntosh-Smith, for his continued support and guidance, and to Wayne Gaudin for all his support and encouragement throughout the entire PhD process — I am indebted to you both.

I'd like to thank my examiners, Revd Dr Jeremy Yates and Dr Wes Armour, for their effort in reviewing my thesis and their kind and considered feedback during the viva.

I would also like to thank Richard Smedley-Stevenson and Dave Barrett for many stimulating discussions about deterministic transport throughout the course of this PhD. And of course thanks must go to my colleagues in the HPC group at Bristol University.

My thanks also go to the careful proof reading efforts of many people, including Wayne, Mum, Richard and Justin. Many typos were fixed as a result of their eagle eyes!

Finally, I would like to thank my family, including Nanny, for all they have done to support me through my (long) education, and Hannah P for sticking with me through the whole thing.

I could not have completed this thesis without all of your help.

Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Access to Piz Daint is thanks to Maria Grazia Giuffreda of CSCS at the Swiss National Supercomputing Centre. Access to the Cray XC40 supercomputer, Swan, and the Cray CS cluster, Falcon, was kindly provided though the Cray Inc. Marketing Partner Network. Access to the Sandia National Laboratories Advanced Systems Technology Test Beds was kindly provided by Simon Hammond. Access to the IBM Power 8 system, Saffron, was kindly provided by Advanced Research Computing at Oxford University. This research used resources thanks to the University of Bristol Intel Parallel Computing Center, including access to Intel compilers and Intel Xeon Phi hardware. This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol — http://www.bris.ac.uk/acrc/. Access to the PGI compiler was thanks to Douglas Miles of PGI. Access to the Edison supercomputer is thanks to Alice Koniges at the National Energy Research Scientific Computing Center. This PhD has been financially supported by the UK Atomic Weapons Establishment.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signed: ____

Date: ____

Contents

List of Figures

List of Tables

List of Listings

Glossary

angular flux The solution of the transport equation describing the movement of neutral particles in the angular, energy and spatial domains, denoted ψ.

cross section Data representing the probability of neutral particles interacting with a material, denoted σ.

discrete ordinates Angular domain discretisation used in the solution of the transport equation, denoted S_n.

grind time The time taken to update one problem unknown in the angular flux calculated as the total application time divided by the product of the iteration count and the problem dimensions.

memory bandwidth Measure of the amount of data that can be moved in a fixed period of time, usually measured in Gigabytes per second.

negative flux fix-up A non-linear routine to ensure non-negative solutions occur, typically by setting negative fluxes to zero and resolving.

octant Group of discrete ordinate angles in 3D space.

pencil The long and thin shaped sub-domain formed via the KBA decomposition scheme.

population The integral of the scalar flux over the spatial domain resulting in a particle count per energy group.

quadrant Group of discrete ordinate angles in 2D space.

quadrature set Angles and associated weights in the angular discretisation; specifically the points used to approximate the integration of the angular flux to form the scalar flux.

scalar flux The integral of the angular flux over the angular domain, denoted ϕ.

streaming-collision operator Part of the transport equation which describes the loss of particles due to streaming (leaving the domain) or by collisions within the material, denoted $\hat{\Omega} \cdot \vec{\nabla} + \sigma$.

sweep Traversal through the spatial domain for an angle respecting the upwind data dependency during solution of the transport equation.

Acronyms

S_n Discrete Ordinates.

API Application Programming Interface.

APU AMD Accelerated Processing Unit.

CPU Central Processing Unit.

CSCS Swiss National Supercomputing Centre.

DAG directed acyclic graph.

DDR double data rate dynamic random-access memory.

DG discontinuous Galerkin.

DRAM dynamic random-access memory.

DSPs digital signal processors.

ECC error correcting code.

FD finite difference.

FEM finite element method.

FLOP floating point operation.

FLOPS/s floating point operations per second.

FPGAs field-programmable gate arrays.

GB 10^9 bytes.

GiB 2^{30} bytes.

GPU Graphics Processing Unit.

HBM High Bandwidth Memory.

HPC High Performance Computing.

KBA Koch, Baker and Alcouffe.

KNC Intel Xeon Phi (Knights Corner).

KNL Intel Xeon Phi (Knights Landing).

LANL Los Alamos National Laboratory.

LAPACK Linear Algebra PACKage.

LLNL Lawrence Livermore National Laboratory.

LU lower upper.

MCDRAM Multi-Channel DRAM.

MiB 2^{20} bytes.

MKL Intel Math Kernel Library.

MPI Message Passing Interface.

NERSC National Energy Research Scientific Computing Center.

NUMA non-uniform memory access.

ORNL Oak Ridge National Laboratory.

PCE Parallel Computational Efficiency.

PCIe Peripheral Component Interconnect Express.

PFLOPS/s 10^{15} floating point operations per second.

QPI Intel Quick Path Interconnect.

SIMD Single Instruction Multiple Data.

SM streaming multiprocessor.

SMT simultaneous multithreading.

SNL Sandia National Laboratory.

SPMD Single Program Multiple Data.

TFLOPS/s 10^{12} floating point operations per second.

UK-MAC United Kingdom Mini-App Consortium.

UVM Unified Virtual Memory.

CHAPTER 1

Introduction

Algorithms with high concurrency are required by many-core technologies at the vector, node and interconnect level if the power of Exascale platforms are to be effectively harnessed for scientific applications. Heterogeneous compute environments, traditional supercomputing nodes enhanced with an accelerator device, are demonstrating high performance per Watt (of required power) with very high thread count compute units, and so therefore form part of a viable roadmap to Exascale. Additionally, the traditional homogeneous compute environments consisting of CPU-style cores are changing, with much increased core and thread counts on each socket. These changes in hardware are disruptive to current algorithms and without research such as that presented in this thesis, the suitability of current algorithms and their implementations may not allow for exploitation of this technology. Such research into mapping them to new architectures is vital for future scientific productivity.

One important scientific computing algorithm is that used to model the movement and interaction of neutral particles, such as neutrons or photons, through materials of varying properties. Example use cases for this model occur in both the fission and fusion nuclear reactor communities, for spectral ocean wave modelling important for weather and climate science, and within the field of bio-medical imaging for treatments such as proton beam therapy and radiotherapy [60, 36, 56]. This behaviour is modelled as an integral-differential Boltzmann balance equation where the solution is highly dimensional, consisting of 3D space, 2D angular direction, 1D energy groups, along with time. Only for simplistic problems can this equation be solved analytically, and therefore the solution of the equation is approximated using numerical methods and performed on large High Performance Computing (HPC) systems. It is estimated that 50–80% of simulation time is devoted to such transport codes on United States Department of Energy supercomputer systems [43]. Therefore a fast and efficient code base to support the solution of this equation is critical.

The widely used implicit numerical schema for solving the transport equation is known as Discrete Ordinates (S_n). It is a tried and tested deterministic method pioneered in the early 1950s by Carlson [19], but the emergence of many-core technology brings a significant challenge to ensure that sufficient levels of concurrency are found so that this method may be efficient on the next generation of architectures.

As with many applications utilising HPC to solve equations numerically, the equation is discretised and the full problem domain split between many discrete computational nodes. The discretisation according to the S_n algorithm imposes data dependencies on the order in which the solution is calculated. Each cell utilises data from upwind neighbouring cells, and calculates outgoing values for consumption by downwind neighbouring cells. This results in a *sweep* through the spatial domain for sets of directional components beginning at each corner of the grid. As such each cell must be computed in a specified order and all cells cannot be computed concurrently in any order unlike many other grid based applications. This is very much in contrast to the typical halo exchange of ghost cells to neighbouring processors seen in many other HPC applications and as such imposes significant challenges. Note too that a solution is sought for every angular direction and energy group, so multiple problem unknowns exist and must be calculated in each cell. In the version of the equation studied in this thesis the angular and energy domains are treated independently for each solve after Baker [13], with appropriate coupling of these domains occurring in the source terms which reside outside of the sweep routine.

The numerical schema results in a deep nesting of iterative loops, where for each timestep a simple iterative scheme is used which is derived from solving the left hand side of the transport equation given an estimate for the right hand side. The solution of the left hand side is found by a matrix-free solve; it is this part which forms the sweep across the spatial domain. This deeply nested loop structure presents many unique challenges.

The high dimensionality of the solution also imposes severe memory footprint requirements on the computing resources. It is common practice to spread the problem across many distributed nodes in order to be able to obtain sufficient memory capacity (similar to strong scaling); this is essentially using the largest possible domain per processor and increasing the processor count until the desired problem size is reached (weak scaling). This decomposition over the spatial domain occurs according to the Koch, Baker and Alcouffe (KBA) algorithm [50]. Rather than decompose a 3D mesh to processors in a fashion to minimise the surface area of the sub-domains in order to reduce the size of the communication thus resulting from a 3D decomposition, a 2D decomposition occurs so that each processor is given the full extent of one spatial dimension in order to minimise the number of idle processors as the sweep progresses over the mesh. The trend in the improvements to memory architectures, primarily in the form of high bandwidth memory, are resulting in improved performance characteristics at the expense of reduced memory capacity in comparison to the traditional DDR memory technologies. Therefore investigating both the scalability of the algorithm as well as approaches to mitigate the capacity constraints are crucial.

With simulation now being a crucial component of modern day science, it is easy for study of a HPC application to become focused on solving particular physics (input) problems. As a result there is often a large amount of legacy code development which has resulted through organic growth into a large, unwieldy code base. It becomes a challenge therefore to investigate the true nature of the performance of the algorithm itself. Accordingly, proxy applications have been developed within the community of both computational and computer scientists. Such proxies are designed to capture the essential behaviour of an algorithm in a small code base, without any of the extra associated components of a production code such as complex mesh generation, I/O and error handling. Also known as mini-apps, they focus on the loop structure, data movement and the appropriate mixture of floating point operations (FLOPs) present in the algorithm, rather than scientifically accurate input and output data, so that the computational nature of the algorithm itself may be captured. This provides an agile vehicle in which to explore the algorithm on modern computer architectures, and a number of mini-apps will be both used and developed as part of this thesis.

Research into sweep algorithms on advanced architectures up until now has focused on more simplistic algorithms, such as matrix factorisation where the findings are not transferable to the sweep which is found in the solution of deterministic transport. On transport solvers specifically, previous work (for example [2, 9, 3]) focused primarily on the scalability of the KBA algorithm (see Chapter 7) with new decomposition schemes designed for the IBM Blue Gene range of supercomputers. Only the work of Villa et al. had examined transport applications on GPUs, however they were unable to show that performance improvements are possible [94]. This thesis shows that in fact GPUs can be leveraged to provide performance improvements.

This thesis therefore takes a holistic approach in order to examine the solution of the transport equation on advanced multi- and many-core architectures. The effects of the data dependency imposed by the sweep through the spatial domain must be understood so that the benefits and characteristics of many-core devices can be exploited successfully. In addition, the scaling challenges presented by the sweep will be characterised, taking account the effects of the interconnect; and in particular account for when the computation is accelerated via the use of many-core technology. The memory requirements put pressure on the entire memory hierarchy, from main memory through to the varying levels of data cache, and these effects too will be explored.

1.1 Contributions

The contributions this thesis offers may be summarised as follows:

- The BabelStream benchmark is developed, which will allow a comparable, cross-platform, cross-vendor, memory bandwidth measurement. This benchmark explores the portability, or lack thereof, of parallel programming models for simple memory bandwidth bound kernels. This gives computer scientists a ‘line in the sand’ (baseline) to compare memory bandwidth bound kernels against; an approach used throughout this thesis to compare the efficiency of attained memory bandwidth of transport kernels.

- A new concurrency scheme for the transport sweep is both developed and successfully implemented on GPU devices, showing that all parallelism in the transport solve needs to be exploited in order to achieve good performance on GPU devices. This scheme achieves significant speedups for the SNAP proxy application from Los Alamos National Laboratory (LANL), the first time such speedups have been demonstrated. The runtime speedups will be shown to be in line with the memory bandwidth advantages of GPU architectures, as measured by BabelStream, and verify that the new concurrency scheme successfully utilises the memory bandwidth for performance.

- Two new transport mini-apps, mega-stream and mega-sweep, are designed and implemented to capture important characteristics of the transport kernels in larger applications such as SNAP. The mini-apps are used to conduct a deep dive study into the cache behaviour of transport on CPU-based processors, and to highlight the complex data reuse pattern in the transport kernel. The work focuses on the restricted cache hierarchy of the Intel Xeon Phi (Knights Landing) (KNL).

- A cross-platform performance model of the scalability of the transport sweeps on multiple nodes of a supercomputer is presented. The model leverages previous work by Bailey and Falgout [9] and extends it for modelling both CPU and GPU architectures. The model was validated at scale on the world's two largest GPU-enabled supercomputers, Titan and Piz Daint. This scaling study shows that the transport sweep becomes network bound at scale, and emphasises the need for high performance interconnects.

- A high order linear discontinuous Galerkin (DG) finite element method (FEM) port of the SNAP mini-app is presented. Whilst unusual to use this method on a structured grid code, it shows that such a high order method may allow for a reduction in memory footprint of a structured grid transport solve; a key challenge as the reduced capacity of High Bandwidth Memory (HBM) and Multi-Channel DRAM (MCDRAM) technologies are beginning to dominate architectural design. This implementation also highlights that higher order methods may not necessarily require more FLOPs overall due to increased memory movement resulting in a similar computational intensity to the standard discretisation approaches. This therefore questions the pursuit of high order methods in order to exploit the ever increasing improvements in floating point operations per second (FLOPS/s), which in and of itself does not necessarily yield good performance without careful thought about the movement of memory.

In summary, these contributions represent a holistic study into the solution of S_n transport for structured grids on advanced architectures. The behaviour of the key kernels are investigated at the node level, and the performance at scale is tested and modelled. Further, issues formed by the memory footprint are investigated at all levels of the memory hierarchy, from main memory through cache.

1.2 Thesis overview

This chapter briefly introduced the motivation for the need to study the performance of solving the transport equation on advanced computer architectures, along with the contributions of this thesis to the field. The remainder of the thesis is structured as follows:

Chapter 2 discusses the current trends in the field of HPC computer architecture. Terminology to describe the scaling of HPC codes on large supercomputers is summarised. Descriptions of various parallel programming models and paradigms are introduced, primarily for use in Chapter 3.

Chapter 3 introduces the BabelStream benchmark and presents memory bandwidth results on a wide variety of hardware. These results show that for many parallel programming models, good performance may be obtained for memory bandwidth bound kernels, rendering the choice of model largely down to personal preference; a good state of affairs for application development in this field. These results provide a baseline performance metric which will be used to quantify the utilisation of effective memory bandwidth of key transport kernels in subsequent chapters.

Chapter 4 gives a brief introduction to the transport equation. The focus is on the computational aspects, specifically the data dependencies imposed by the discretisation and the methods of numerical solution. Whilst there is much interest in the derivation or physical ramifications of this equation, the interested reader is referred to other texts on this matter as the focus of this work is on the solution on advanced architectures. The SNAP proxy application is introduced which will provide the vehicle for much of the work presented in this thesis. Other transport proxy applications and their associated research are surveyed, along with summaries of other relevant literature looking into solving sweep based codes.

Chapter 5 describes the parallel scheme of the original SNAP proxy application and introduces the concurrency scheme to enable good performance of transport codes on GPU architectures. The performance of the scheme is tested on a number of GPU devices, and is shown to leverage memory bandwidth advantages of GPU devices. The memory bandwidth of the transport kernel is modelled and compared to results of the BabelStream benchmark in order to present the sustained memory bandwidth of the transport kernel.

Chapter 6 introduces the need for new mini-apps to explore the complex data access patterns in the transport kernel on more traditional CPU-based architectures. The mega-stream and mega-sweep mini-apps are described and baseline performance results are presented. Cache focused optimisations are shown for mega-stream. The chapter pays particular attention to exploiting the cache hierarchy in many-core CPUs, using KNL as its motivation as an exemplar of a cache-based architecture with particular restrictive properties.

Chapter 7 explores the scalability of the transport sweep. A survey of published performance models is presented. One model is enhanced to describe the performance of the sweep in SNAP on both CPU and GPU architectures. This model is verified at large scale on the two largest GPU enabled supercomputers. These weak and strong scaling studies show that the transport sweep becomes network bound at scale.

Chapter 8 includes a discussion of the computational nature of the DG FEM method in comparison with the finite difference (FD) method. A parallel implementation of a FEM discretisation for the SNAP proxy application is described and performance results are presented along with a discussion on memory capacity considerations.

Chapter 9 concludes the thesis with a summary of findings. Avenues for future exploration in this field are presented focusing on extending this work to unstructured domains.

Appendix A introduces the FD method and applies this discretisation approach to the spatial dimensions of the transport equation.

Appendix B introduces the FEM itself and applies the linear DG FEM in order to discretise the spatial dimensions of the transport equation.

CHAPTER 2

High Performance Computing trends towards Exascale

Typical supercomputers of today are constructed from a collection of individual compute nodes linked via a network interconnect. The nodes contain one or more CPU processor sockets, and may also contain some heterogeneous hardware in the form of an accelerator such as a GPU. The GPUs are typically connected via PCIe, although recent advancements have introduced alternatives such as NVLink which is only available for certain combinations of CPU and GPU. The hardware itself although often being High Performance Computing (HPC) focused is made from commodity technology; this is in contrast to specialist hardware such as Anton or early Cray systems. Zivanovic et al. surveyed the memory capacities of the supercomputers in the Top 500 list and found that most CPU based nodes are constructed with 2–3 GB of memory per core [98]. For GPU based nodes the memory capacity of the GPU is set by the GPU vendor and is not a choice during system construction; each GPU however typically contains (up to an order of magnitude) less memory than the CPU host node. The addition of heterogeneity into the computing node along with increased core count in standard CPU architectures is causing significant disruption for both system design and algorithms and applications.

The network interconnects however are typically designed for high performance rather than using standard technology such as Ethernet. All the interconnects focus on low latency and high bandwidth connections between nodes. Examples include InfiniBand, Intel TrueScale and OmniPath, Cray Gemini and Aries and custom interconnects such as those in Tianhe-2. The network connections are made to improve communication times between nodes, and the design of this is known as the network topology. For example, the 3D torus topology used in the Titan supercomputer connects nodes together in a 3D mesh. The network allows neighbouring nodes to communicate efficiently, however long range connections may require a high number of network hops. The Dragonfly topology used in the Piz Daint supercomputer on the other hand connects islands of nodes with all-to-all connections, with nodes inside the group connected using another topology. Therefore long range communication aims to be min-

	Titan	Piz Daint (2012–16)
Machine	Cray XK7	Cray XC30
Processor	AMD Opteron 6274	Intel Xeon E5-2670
Cores/processor	16	8
GPUs	NVIDIA K20X	NVIDIA K20X
Processors/node	1	1
GPUs/node	1	1
Nodes	18,688	5,272
RAM/node	32 GB + 6 GB	32 GB + 6 GB
Interconnect	Gemini (3D torus)	Aries (Dragonfly)
RMAX [89]	17.6 PFLOPS/s	6.3 PFLOPS/s

Table 2.1: The Titan and Piz Daint supercomputers

imised as all islands can communicate directly, whilst neighbour communication within the island is also efficient.

Information about Titan and Piz Daint, the two supercomputers used heavily in this thesis are detailed in Table 2.1 as an example of modern supercomputer construction.

This chapter will discuss some current trends in HPC architecture and introduce some important concepts which will be used throughout the remainder of the thesis. Formal definitions of scaling will be presented for use in analysing the performance of transport applications on a large number of compute nodes. The applications used and written for use in this thesis are written in a number of different parallel programming models, and summaries of them are to be found in this chapter. Important concepts for modern multi-core CPUs such as non-uniform memory access (NUMA) and vectorisation will also be defined so that they can be used without further explanation in the remainder of the thesis.

2.1 Moore's law

In 1965, Moore predicted that the number of components in integrated circuits would double every year [74]. Moore revised this estimate in 1975 to that of doubling every two years [73]. These estimates have come to be known as *Moore's law.*

It is important to note that these laws are expressed in terms of complexity of the circuit, and usually means the number of components. Improvements in lithography are allowing a greater number of (smaller) transistors to be produced in a single chip, thereby increasing their complexity under Moore's law. Dennard scaling refers to transistor energy being proportional to their size. It does not however necessarily refer to the shrinking of transistor size which was the main driver of Moore's law until Dennard scaling drew to an end.

In more recent times, Moore's law continues to hold however the number of transistors has slowed to doubling every three years. The increase in transistor count allows for an increased complexity of the chip. For example, each chip boasts a number of independent compute cores, with each core consisting of vector units.

The Top 500 list records the historical trends in supercomputer design by focusing on floating point performance [91], and McCalpin has also tracked the historical trends by focusing on all aspects of the system including the compute performance, network and memory [68]. The trends are showing that accelerated architectures such as GPUs are becoming more prevalent in the higher echelons of Top 500 supercomputers as a means to continue to offer improved performance over older systems, and therefore ensuring that applications can exploit this hardware successfully will be key for future proofing code. Each aspect of computer architecture has developed at differing paces, and importantly floating point performance has improved much faster than memory technology. McCalpin defines a metric of *system balance*, the ratio of performance metrics, and this highlights that over the last five years floating point performance has increased twice as fast as memory bandwidth, and over the last three years twice as fast as memory latency [68]. The latest computer architectures are providing nearly 100 floating point operations (FLOPs) per word of memory accessed. Unfortunately, many HPC applications including the solution of the transport equation are in fact bound by the memory system of a processor rather than its floating point operations per second (FLOPS/s). It is the novel and disruptive technologies, such as GPUs, which are providing increased memory bandwidths and exploiting this technology will be key to continued performance gains with future systems.

2.2 Scaling

The relationships between the runtime of an application and the number of processors used to execute the application can be captured by a number of metrics which are known as forms of *scaling*. These metrics give a way to evaluate the efficiency of the parallel program, and may guide appropriate maximum processor counts for application runs.

2.2.1 Amdahl's law

Amdahl observed that if a portion of an application is not able to utilise additional processors then the overall speedup that is possible is limited [6]. This can be represented in the form of a numerical rule as presented by Gustafson [39]:

$$S_A = \frac{s+p}{s+\frac{p}{n}}$$

which describes the available speedup, S_A, that is possible from n processors for an application with a runtime of the sum of the serial, s, and parallelisable, p, portions. As an example, for an application where 90% is run in parallel, the maximum possible speedup *of the entire code* on 1000 processors is only 9.9X faster than totally serial execution; the serial portion of the code becomes dominant in the overall runtime. If that application was improved so that 99.9% was run in parallel, the maximum possible speedup of 1000 processors would be 500X faster than serial execution. Therefore the fraction of serial code may become dominant at a large number of processors.

2.2.2 Strong scaling

Strong scaling refers to the speedups that can be achieved from running an application with a *fixed* problem size on multiple processors. With a fixed problem size, as the problem is decomposed across the processors, each processor will receive less work as more processors are added. The formula may be derived from a simplification of Amdahl's law:

$$S_s = \frac{T_1}{T_n}$$

where T_1 is the serial execution time ($T_1 = s + p$) and T_n is the execution time on n processors ($T_n = s + p/n$). For *perfect* strong scaling one would expect a linear speedup of n times (nX) on n processors; for example if the number of processors doubles the runtime should be halved resulting in a speedup of 2X. However, serial portions of the code as a result of Amdahl's law, along with any overheads of running at large processor counts with diminishing local problem sizes may be causes of imperfect strong scaling. Additionally the manner in which work is shared between processors may be inherently unable to scale linearly with processor count – it is such an application which is the subject of this thesis.

2.2.3 Gustafson's law

Whilst extra computational resource may be used to reduce the overall runtime of an application through strong scaling, this extra processing power may be used alternatively to increase the size of problem to be solved. Gustafson's law describes the available speedup where the problem size is scaled with the number of processors [39]:

$$S_G = \frac{s + p \times n}{s + p}$$

whereby unlike in Amdahl's law, $s + p$ represents the *parallel* execution time on n processors. Therefore the numerator in this speedup metric would be the time taken by a serial processor to execute the global problem; interestingly this assumes that the parallel part of the problem is able to perfectly strong scale out to n processors.

2.2.4 Weak scaling

Weak scaling refers to the speedups achieved from running an application with a fixed problem size *per* processor. Therefore the global problem size is usually the product of the number of processors and the size of the problem on each processor. The formula can be derived from a simplification of Gustafson's law:

$$S_w = \frac{T_1 \times n}{T_n}$$

where T_1 is the serial execution time of the global problem and T_n is the execution time on n processors. A *perfect* weak scaling would result in a speedup of 1, whereby the runtime is constant. The addition of processors does not increase the overall runtime thereby solving a larger problem in the same runtime.

2.2.5 Parallel efficiency

One can use the strong and weak scaling models to construct a simple efficiency metric:

$$\text{PE} = \frac{S}{n}$$

where either of the strong (S_s) or weak (S_w) scaling speedup formulas can be used to get strong or weak scaling efficiency respectively. For convenience this fraction is often converted into a percentage.

Specifically therefore, strong scaling efficiency can be calculated as:

$$\text{PE}_s = \frac{T_1}{T_n \times n}$$

with weak scaling efficiency calculated as:

$$\text{PE}_w = \frac{T_1}{T_n}$$

In both cases an efficiency of 100% would indicate perfect scaling.

2.3 Programming models

Parallel programming models are an enhancement to a standard programming language to express parallel execution of code statements. They often come with both a memory and execution model in order to determine synchronisation requirements, in particular formalising data sharing between parallel units. They allow a programmer to write code which can run on multiple hardware execution units, which may be present in a single hardware resource (such as cores) and/or physically separated resources (such as compute nodes on a network). They often take the form of compiler directives, language extensions or an Application Programming Interface (API).

2.3.1 Message Passing Interface

Message Passing Interface (MPI) defines a standard way to communicate data between processes via a network [71]. In principal, the computers are set up in a communication group, and messages can be sent in a point-to-point or broadcast manner between nodes in the group. Originally these consisted of single processor nodes connected together on a network. The nodes do not share memory, and the only way that data from one node can be read by another node is by the first node sending a copy of that data over the network. This is known as a distributed memory model.

With the advent of multi-core processors, and dual-socket nodes, more than one Message Passing Interface (MPI) process may be run on any one physical compute node; any memory allocated by each process is inaccessible by another process. As such MPI can also be used to exploit the parallelism at the node level. This leads to the term *flat MPI*, where MPI is used to describe the parallelism at both the node and the interconnect level, with an instance of the program running on every core of every node. Messages sent within a node tend to exploit locality knowledge and use the shared memory space, rather than communicating via a network interface; however this is a detail of the implementation of the MPI runtime.

MPI can be combined with a shared memory parallel programming model to explicitly describe the node level and interconnect level parallelism; all the models listed in the upcoming sections are compatible. This is typically called a *hybrid* model and denoted MPI+X, where X is the other 'within node' parallel programming model. The correct balance of flat MPI or varying degrees of hybrid MPI is difficult to predict in the general case due to communication costs, memory footprint and NUMA effects of hardware cores within a shared memory node.

2.3.2 OpenMP

The OpenMP API is a collection of compiler directives and standardised subroutines. The parallel paradigm is that of fork-join, where the starting serial execution later splits into parallel threads (fork) which are then combined back to serial execution after synchronisation (join). These parallel threads operate redundantly in regions of code marked using the compiler directives, unless a work-sharing construct is also specified. The work-sharing constructs are typically applied to loops where the iterations may be run in parallel, and by specifying the work-sharing construct the iterations of the loop are shared between the available threads.

OpenMP utilises a shared memory model, where the memory is available to all threads. As such it is the responsibility of the programmer to ensure data races are not introduced and appropriate synchronisation takes place. To assist, OpenMP includes support for specifying atomic operations or critical regions to ensure only one thread modifies the data at a time.

Initially OpenMP was specified with multi-core CPU architectures in mind, but the OpenMP 4 standard introduced the concept of offload to an attached heterogeneous compute device with its own memory space. The memory model is therefore expanded to support movement of memory between the host and device memory spaces. Much of the memory movement is handled automatically by the OpenMP implementation, however arrays allocated on the heap in particular must be moved manually between the two memory spaces using the constructs in the OpenMP API. In a similar manner to the thread model, structured blocks of code are annotated with compiler directives to state that they should be executed on the device. By default the offload occurs in a blocking manner where the host waits until the offload is complete.

The OpenMP 4.5 refinement affords the programmer greater control of memory movement by allowing them to move memory using compiler directives in an unstructured manner, rather than enforcing a dependency on scope (through structured code blocks). This allows for much more flexibility in the development of applications and prevents invasive and disruptive code changes.

2.3.3 OpenACC

The OpenACC programming model began life as an investigatory project to inform the OpenMP standard as to the best ways to augment the specification with device offload support. Subsequently, OpenACC has been marketed strongly by its creators PGI and their parent company NVIDIA and today retains enough momentum for it to be considered in its own right.

Compiler directives are added to simple loops to highlight to the compiler that they may be parallelised. The compiler is then free to parallelise this code as it sees fit through the use of the descriptive `kernels` directive, with the specification describing this as creating 'kernels' for offload — a recursive definition but taken to mean that the compiler can do whatever it thinks best utilising auto-parallelisation research. The creators of OpenACC, PGI, advocate using this mode of execution [72].

Some more control is given to the programmer through an additional `parallel` directive. In this mode loops are annotated to say that the iterations are safe to be computed in parallel and to be distributed to threads; in OpenACC a thread refers to a vector lane.

The device memory space must also be explicitly managed by the programmer through additional compiler directives.

Full support of OpenACC is limited to a single compiler, owned by PGI, and as such the target devices are limited to NVIDIA GPUs, Power 8 CPUs and x86 CPUs. The Cray and GCC compilers also provide some OpenACC support.

Unlike with OpenMP 4.5, the offloaded code is by default non-blocking, and the host is free to continue working; the programmer is able to specify otherwise.

OpenACC allows memory to be cached in the scratchpad memory available on GPUs, although the functionality is rather limited. Only slices of already allocated arrays may be copied in, and there are many restrictions around this. At the time of writing, we have confirmed after private communications with PGI developers that the standard approaches for tiling matrix multiplication operations are not able to be implemented correctly in the programming model whilst also providing performance.

2.3.4 CUDA

The proprietary CUDA API provided by NVIDIA is designed for programming their GPU devices using a C/C++ based language. The device code is written as function calls in a single source code base. Much of the complexity of obtaining a GPU device is removed through the use of default behaviour, however control is provided for more complex situations. The functions to run on the device are decorated with attributes and launched specifying launch parameters with chevron notation. Logical threads are grouped together into thread blocks, and the launch parameters determine the number of threads per block and the number of blocks launched on the device; these parameters must be specified for every kernel call. The physical GPU itself is constructed from a number of streaming multiprocessors (SMs) each consisting of a number of CUDA cores.

The memory of the device is typically a separate memory space, although in modern versions of CUDA the address space can be shared between the host and the device, and the same pointers may be used everywhere, although the programmer is responsible for synchronisation. CUDA can also take total control of memory movement, and therefore explicit movement need not necessarily be written by the programmer. This is taken further in CUDA 8 along with the Pascal GPU architecture which is able to page fault and migrate and cache memory on the device on demand.

The kernel functions are typically written in a way that encapsulates the loop body, with the launch parameters on kernel execution specifying the loop iteration extent. As such the code can look different to the compiler directive based approaches used in some programming models such as OpenMP and OpenACC where the loop itself is still written in the source code.

2.3.5 OpenCL

Khronos released OpenCL in 2009 as a royalty-free, open standard API. The model is designed for programming an attached accelerator device which is constructed out of processing elements grouped into compute units. This is a generic model and can be applied to a range of devices including CPUs, GPUs, field-programmable gate arrays (FPGAs) and digital signal processors (DSPs). On a CPU, a compute unit is typically seen as a core and a processing element is a vector lane.

The device has its own memory space and memory movement must be explicitly managed by the host program (later versions of OpenCL introduced a shared memory model whereby the movement of memory could be handled by the framework itself).

Kernels are written in plain text in a subset of C99 and compiled at runtime to allow execution on the device. A command queue is used to order the kernels, memory transfers and synchronisation between the host and device.

Work-items characterise single instantiations of the kernels, and are logically organised into work-groups, which can be one to three dimensional. On-device synchronisation is allowed between work-items within a single work-group, but not between different work-groups; this synchronisation occurs at the end of the kernel execution.

2.3.6 Kokkos

Kokkos is a C++ abstraction layer, allowing programmers to write computational loop bodies as lambda functions, and execute them on a variety of architectures through a number of frameworks, such as Pthreads, OpenMP and CUDA. It is developed by Sandia National Laboratory (SNL) and is available as open source. Data structures are encapsulated into Kokkos Views which handle the allocation of memory in the correct way for each device. The Views differentiate between host and device memory spaces in a consistent way. Execution of the code on a different device therefore only requires changing the target device and recompiling the application. The lambda functions are executed with a parallel dispatch function, which may be nested to express hierarchical parallelism. The Kokkos implementation deals with the mapping of parallel execution onto one of the backend models.

2.3.7 RAJA

RAJA is another C++ abstraction layer which can target CPUs via OpenMP and NVIDIA GPUs via CUDA. It was released by Lawrence Livermore National Laboratory (LLNL). Execution policies describe how the iteration space may be executed in parallel. Memory allocations are also linked to a policy so that the ordering of data in memory can match the appropriate execution order for good performance. This allows the implementation of tiling schemes to be simplified, as the correct data allocation is also taken care of by the abstraction layer. The computational kernels are expressed as lambda functions and run via a parallel dispatch function. The policies and target devices are typically described once for each loop bound in a single source location so it is simple to update these for execution on a different device.

2.3.8 SYCL

The SYCL C++ abstraction layer was developed by Khronos over OpenCL, allowing for single-source codes entirely written in C++ unlike with OpenCL which is C based and requires separate kernel source. Like OpenCL, SYCL is an open standard. The kernels are written as lambda functions. SYCL is designed to be close to C++14 so that a standard compiler may build it along with a header and run on a CPU, with SYCL device compilers providing the infrastructure for running on other devices which support OpenCL. The full machinery of OpenCL is also exposed via the SYCL API if the programmer requires it. The execution model is the same as OpenCL, consisting of work-items and work-groups.

2.4 Vectorisation

Instructions generally operate on a single numerical value at a time, such as adding two numbers together. Vector instructions increase the number of arithmetic operations possible (of floating point, integer, etc. data types) by performing a single operation (instruction) on more than one data item, but only requiring a single instruction to be issued. As the number of instructions is reduced this can also alleviate pressure on the fetch-decode units of the instruction pipeline. The vector width is often chosen to match the width of crucial data paths, often those to the cache hierarchy. These instructions fall into the Single Instruction Multiple Data (SIMD) classification of execution, where a single instruction is issued which causes an operation to be performed on multiple data items. This is in contrast to Single Instruction Single Data execution where an instruction only operates on a single data item. In x86 architectures, vector registers are provided by the hardware which hold multiple data items.

Many optimising compilers are able to generate vector instructions in the instruction stream. This is made possible through loop analysis which examines data dependencies between memory accesses in the source code, and where no dependencies occur vector instructions may be generated. Many heuristics are used to balance the potential improvements to runtime that vector operations provide with any overhead of performing the vector instructions. Writing source code in order to take advantage of this compiler technology therefore requires some care. Compilers may also provide programmers with a number of compiler directives in order to direct the dependency analysis, and provide additional details about loop trip counts, data alignment and dependencies. Some of these directives are also available in OpenMP 4.5.

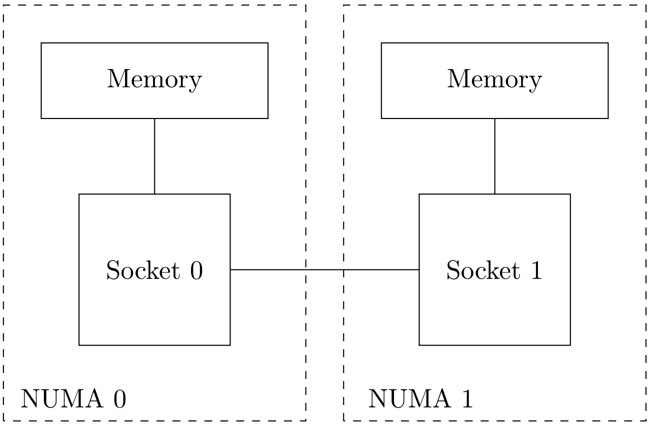


Figure 2.1: Example NUMA regions in a dual-socket node

2.5 Non-uniform memory access

A typical two socket CPU node will usually provide two NUMA regions, each assigned to a socket, with a shared address space. Memory access timing is uniform within the region, but access to a different region takes much longer.

Two NUMA regions for an exemplar dual-socket CPU system are shown in Figure 2.1. The sockets are connected via an interconnect such as the Intel Quick Path Interconnect (QPI). Each socket is connected to some memory; this memory is mapped as a single address space and is therefore accessible by cores in each socket. However, the memory space is partitioned into NUMA regions, so that access to the memory has different latencies depending on the path taken. For example, it is fastest for a core in the socket to access only memory associated with the same NUMA region; thus for Socket 0 it is fastest to access memory directly connected to its own memory controllers, the memory residing in NUMA region 0. This same address is accessible from Socket 1, however the data must also travel through the socket-to-socket interconnect which increases the latency of memory access.

Memory is typically allocated according to a first-touch policy. When a memory address is first accessed by a particular core residing in a NUMA region, a page is allocated in the memory also associated with that same NUMA region. Therefore for performant memory access in computational kernels, memory should have been previously initialised in the same NUMA region that the access takes place in.

2.6 Directed acyclic graphs

A directed acyclic graph (DAG) is a data structure which describes the connectivity between a set of vertices (or nodes). This is typically notated $G = (V, E)$ where the graph G consists of a set of vertices V and edges E. The vertices are connected together with edges, where an edge (i, j) specifies that there is a connection from vertex v_i to vertex v_j. As the graph is directed, an edge only describes a one-way relationship, and so does not also specify an edge exists in the reverse direction, i.e. (j, i). The graph is also said to be acyclic is

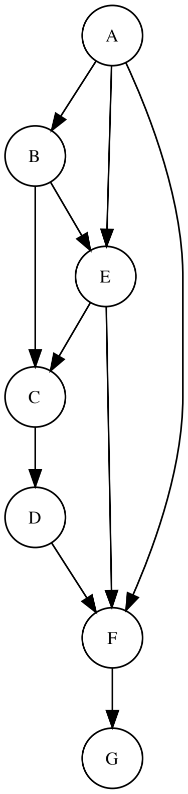


Figure 2.2: An example DAG

there are no cycles in any path through the graph; that is a path through the graph following the edges does not go through the same vertex twice.

An example directed acyclic graph (DAG) is show in Figure 2.2. This graph consists of seven nodes and ten directed edges. It is also an example of an acyclic graph.

Although the study of graph properties is a field in its own right, their relevance to this thesis is in their description of the scheduling of concurrent work. In the mesh based approach of the deterministic transport application, the graph vertices are cells in the mesh with the graph edges describing the dependencies between the cells.

2.7 Summary

This chapter sets the scene of the current and near-future landscape of HPC. The number of cores in a multi-core CPU are increasing, with NUMA effects and vectorisation also becoming important to consider. More advanced computer architectures such as GPUs are requiring increased levels of concurrency in the algorithm but do provide memory bandwidth improvements. This trend is set to continue on the path towards Exascale systems. Therefore it is important that all aspects of the HPC infrastructure, including programming models and applications, are able to explore and exploit these different and disruptive technologies.

CHAPTER 3

Measuring achievable memory bandwidth across diverse many-core architectures

The work in this chapter also appears in the following publications:

- Tom Deakin and Simon McIntosh-Smith. *GPU-STREAM: Benchmarking the Achievable Memory Bandwidth of Graphics Processing Units (poster).* IEEE/ACM Supercomputing, 2015.
- Tom Deakin, James Price, Matt Martineau and Simon McIntosh-Smith. *GPU-STREAM: Now in 2D! (poster).* IEEE/ACM Supercomputing, 2016.
- Tom Deakin, James Price, Matt Martineau and Simon McIntosh-Smith. *GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models.* Performance Portable Programming Models Workshop at International Conference on High Performance Computing, 2016.
- Karthik Raman, Tom Deakin, James Price and Simon McIntosh-Smith. *Improving Achieved Memory Bandwidth from C++ Codes on Intel Xeon Phi Processor (Knights Landing).* The Intel Xeon Phi Users Group Spring Meeting, 2017.
- Tom Deakin, James Price, Matt Martineau and Simon McIntosh-Smith. *Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream.* International Journal of Computational Science and Engineering (special issue, in press), 2017.

A great many optimised High Performance Computing (HPC) applications move a lot of memory in comparison to the number of floating point operations (FLOPs) and as such their performance becomes limited by the available memory bandwidth of any particular hardware. The need to quantify what the achievable performance of a memory bandwidth bound code could be has led to the development of the gold standard of memory bandwidth benchmarks: STREAM [66]. Indeed, the STREAM benchmark may be considered one of the earliest mini-apps due to its development by McCalpin to consider the expected performance of Earth Science codes on differing architectures [68].

The solution of the transport equation which is the focus of this thesis is an example of a code which has relatively few FLOPs compared to the number of memory accesses, and as such its performance should be bound by the memory architecture. By first measuring the achievable memory bandwidth for a simple routine it is then possible to quantify the percentage of bandwidth that a more realistic kernel achieves. The STREAM benchmark, although prevalent in the HPC community, has only been ported to very few programming models, and for a limited range of computer architectures; in particular it does not run on GPUs. Therefore the BabelStream benchmark (formally known as GPU-STREAM) was written in order to quantify the memory bandwidth of a range of devices in a portable, and importantly, fairly comparable manner. This chapter addresses findings for BabelStream in its own right, but the results will become the backbone of the comparative performance analysis of later chapters.

3.1 Memory hierarchy

In a well optimised high performance application, one would expect that a particular feature of the computational hardware is being saturated such that it becomes the limiting factor for performance. Traditionally this was typically the floating point computation itself. The limit was how fast basic arithmetic of real numbers could be calculated. As such the LINPACK benchmark was developed to compare different supercomputers in terms of their computational weight for arithmetic [30]. The benchmark produces a single number with units of floating point operations per second (FLOPS/s). Therefore if a machine with a higher FLOPS/s rate is developed you could therefore run your own scientific application faster there. One could even make rough estimates at the potential improvements, where a doubling of the FLOPS/s rate could mean runtime halving. The LINPACK benchmark provides sets of historical data via the Top 500 lists in order to track this improvement [90].

However over time the limiting factor for many applications is no longer the arithmetic but rather data movement. Data caches are used to store portions of data from main memory close to the execution units in the processor in order to improve access times for frequently accessed data. Figure 3.1 shows the approximate number of cycles to access each level of the cache hierarchy, using data from Intel processors [47, 35]; this is based on the Skylake architecture. In these Xeon processors, each core has its own L1 and L2 cache, with all cores on a socket sharing a L3 cache. The L1 cache is closest to the functional units and has the lowest latency, with latency increasing as memory is further away from the core. Note that latency to main memory is presented in nanoseconds rather than cycles as main memory typically operates on a different clock to the cores; however a cycle time is shown relative to the core clock to give the reader an idea of the number of core cycles which elapse during this time period. With many floating point operations taking just a single cycle it is clear that memory movement is a relatively expensive operation as it takes much longer

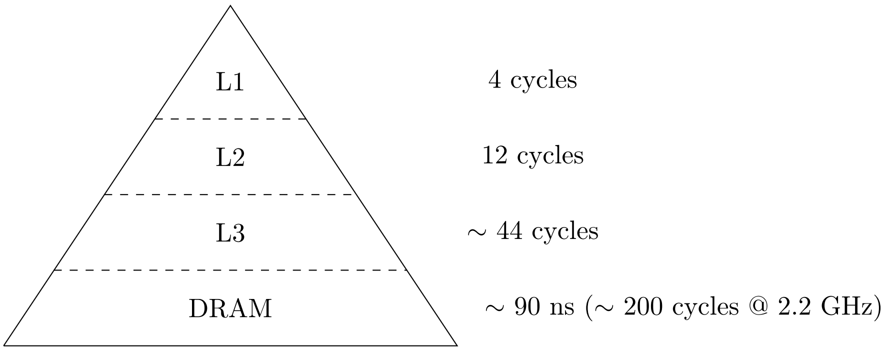


Figure 3.1: Memory access times for levels of the cache hierarchy for Skylake according to Intel and De Gelas and Cutress [47, 35]

.

to move data into the floating point unit than it takes to do the arithmetic once it has arrived. Therefore the cost of data movement should become the focus of optimisation for codes with relatively few FLOPs.

The `lat_mem_rd` benchmark from LMbench can be used to quantify the memory latency associated with memory accesses in the cache hierarchy levels [70]. The results from the benchmark for Broadwell and Intel Xeon Phi (Knights Landing) (KNL) are shown in Figure 3.2; the benchmark was run with 256 MiB arrays with a stride of 256, running 10 repetitions. The Broadwell processor operated at 2.2 GHz and the KNL processor operated at 1.3 GHz. Each level of the cache hierarchy can be seen as horizontal plateaus in the graph with their position on the y-axis denoting the latency. For example, L3 access on Broadwell takes around 17 ns, or around 38 cycles which corresponds to the data shown in Figure 3.1; for this processor a cycle is 0.45 ns. On the KNL there are only three horizontal regions corresponding to the L1 and L2 caches and Multi-Channel DRAM (MCDRAM); note the lack of a L3 cache, and therefore the penalty for missing in L2 is severe. Whilst the KNL latency in elapsed time is generally longer than Broadwell recall that the clock speeds are somewhat different and therefore the number of core cycles per memory access is similar on both architectures. It is the realised latency in core cycles which is the important factor when considering the performance of the memory architecture rather than the real latency in nanoseconds; for this gives an indication into the balance of floating point operation (FLOP) and memory performance of an architecture.

The sizes of the caches are also visible in Figure 3.2; both processors have 32 KiB L1 caches, Broadwell has 256 KiB L2 cache with KNL having 1 MiB although this is shared between two cores on a tile (the benchmark is serial so the full 1 MiB is available to the core), and finally Broadwell has 55 MiB L3 cache. The final plateau corresponds to main memory, which is double data rate dynamic random-access memory (DDR) on Broadwell and MCDRAM on KNL.

McCalpin has analysed the ratio of floating point operations to memory latency (the time to access memory) and found that recent CPU systems can perform around 100 FLOPs whilst waiting for a byte to arrive from the main memory [68]. By examining the historical trends of this ratio McCalpin also

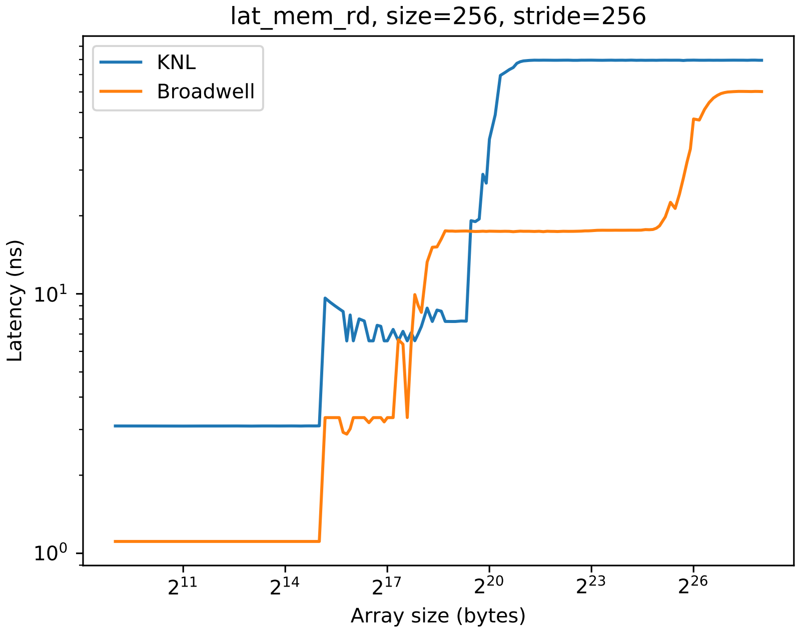


Figure 3.2: Memory latencies on Broadwell and KNL as measured by `lat_mem_rd`

found that memory latency is increasing at around 4% per year, primarily dominated by coherence rather than data transfer combined with decreasing clock frequencies to maintaining strict power (and cooling) budgets. Whilst the peak FLOPS/s rate can be increased through Single Instruction Multiple Data (SIMD) instructions, clock speeds and core count, which are all ultimately driven by improvements in transistor density according to Moore's Law, improvements to the memory system are somewhat more challenging. In particular whilst improvements to memory bandwidth through memory technologies such as MCDRAM and High Bandwidth Memory (HBM) have improved the memory bandwidth, the ratio of FLOPS/s to peak memory bandwidth has still increased at around 14% a year. As such the system imbalance is increasing over time, and for applications with runtime dominated by properties of the memory system each generation of technology provides less improvement than for applications dominated by floating point operations.

3.1.1 The Roofline model

The Roofline model along with refinements such as the cache-aware Roofline model gives a way of categorising what the limiting architectural factor is for important routines, or kernels, in applications based on the ratio of floating point operations to memory accesses [95, 46]. The cache-aware Roofline model is rather more helpful in such discussions as it exposes the full cache hierarchy.

Each kernel is modelled by counting the number of floating point operations and memory load and stores in bytes; the memory operations are counted

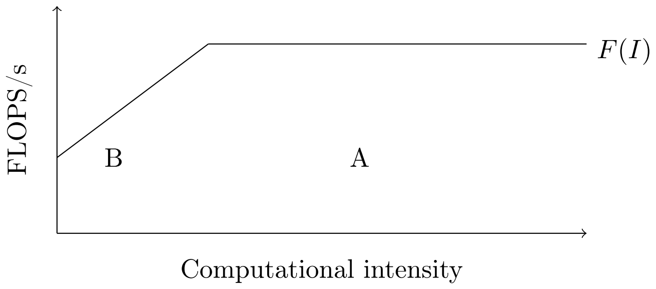


Figure 3.3: Illustration of the cache-aware Roofline model

from the perspective of the kernel so are actually cache-oblivious, as all memory operations are counted the same no matter where they come from. The *computational intensity* is therefore the ratio of FLOPs in the application kernel to bytes moved by the kernel. A ratio of 1 implies one FLOP for each byte moved, with ratios greater than 1 implying more FLOPs than memory movement. Of more interest to this thesis is the case where the ratio is less than 1, with fewer FLOPs than memory moved. One does not have to look particularly hard for simple mathematical functions where this is the case, a simple addition operation requires two numbers for a single floating point operation; and such numbers are typically each represented by 8 bytes in double precision — the computational intensity of this addition is therefore 1/16 excluding writing the result. Other operations require many more FLOPs such as trigonometric functions and square root operations.

The Roofline model relates the computational intensity to peak floating point performance of a particular architecture via the following function taken from Ilic et al. [46]:

$$F_\alpha(I) = \min\left(B(\beta) \times I, F(\phi)\right)$$

where I is the computational intensity, $F(\phi)$ is the peak FLOPS/s of the device and $B(\beta)$ is the peak memory bandwidth available from a particular level of the cache hierarchy. This function can then be plotted on a chart such as is sketched in Figure 3.3. Here the function represents a literal Roofline on achievable performance for a given architecture.

The computational intensity of scientific application kernels can then be modelled and plotted on the Roofline model in order to advise the programmer as to the performance limiting factors. For example, kernel A on the sketch is beneath the horizontal portion indicating that the limiting factor should be floating point operations and even if the memory bandwidth of the architecture could be improved the runtime of the kernel would change little. Whereas kernel B is beneath the diagonal implying that memory bandwidth should be the limiting factor, and increasing the floating point performance of the architecture would not assist. Such guidance is given to the programmer as to which optimisations will give the most beneficial effect. Neither of the kernels A and B are lying on the Roofline itself and so therefore show need for optimisation. For kernel A the focus should be on optimising FLOPs and optimisations to kernel B should prioritise memory movement. A single Roofline plot such as this may include multiple ‘roofs’, each corresponding to a level in the memory hierarchy.

3.2 The STREAM benchmark

Memory bandwidth is the amount of data that can be moved through the system in a given period of time. The memory latency is the time it takes for a data item (a single byte) to move through the system. As such the two quantities are related, but the general approach for improving the speed at which memory is available is to provide a wider interface, or more memory controllers, so that more data can be moved concurrently and therefore more data moved in the time period, increasing the memory bandwidth rather than reducing the latency.

The STREAM benchmark, similar in community adoption to LINPACK, seeks to determine the available memory bandwidth on a particular computational node [66]. The benchmark is made up of simple routines where it is straightforward to model the effective memory bandwidth. The effective memory bandwidth is counted so that each data item is loaded or stored only once in each kernel, and that the data set is large enough that it must be reloaded from main memory (and not cached) when used again in the proceeding kernel. It is constructed from four simple routines utilising simple element-wise arithmetic on large arrays:

1. Copy: $c(i) = a(i)$
2. Multiply: $b(i) = \alpha c(i)$
3. Add: $c(i) = a(i) + b(i)$
4. Triad: $a(i) = b(i) + \alpha c(i)$

In these routines α is a fixed scalar constant.

It is often convenient to speak of the routines in terms of vectors from the field of linear algebra. The Copy operation duplicates the vector in memory. The Multiply operation scales a vector by a fixed quantity. The Add operation performs the element-wise sum of two vectors. The Triad kernel is a combination of Multiply and Add, whereby a vector is scaled and added to a vector with the result stored in a third. Note this is slightly different to the `axpy` routine from Level-1 BLAS where the result is overwritten into one of the vectors: $b(i) = b(i) + \alpha c(i)$.

These routines can be extended by a fifth, from the STREAM2 benchmark [67]:

5. Dot: $\sum_i a(i) \times b(i)$

The Dot routine is a vector dot-product of two vectors and produces a single scalar value.

The STREAM2 benchmark was written to test the different levels of the cache hierarchy. It is a serial code with loops partially unrolled, a technique which for vector machines was designed to assist vectorisation, however on modern SIMD architectures is not recommended as this may generate gather instructions. The number of iterations is different for each kernel in this benchmark, and the control flow is fairly complex. As such this benchmark is not representative of how, for example, a dot-product would be conducted in a larger code today. When comparing to a baseline dot-product performance therefore a simple dot-product kernel in the STREAM regime of fixed iteration counts has been implemented.

Kernel	Load	Store	Total	Computational Intensity
Copy	N	N	$2N$	0
Multiply	N	N	$2N$	1/16
Add	$2N$	N	$3N$	1/24
Triad	$2N$	N	$3N$	2/24
Dot	$2N$	0	$2N$	2/16

Table 3.1: Memory movement in the STREAM kernels

Table 3.1 shows the number of load and store operations required for each routine (kernel) given the number of elements in the array N. It is assumed that N is large enough that the scalar load of the α value is not important. It is simple to calculate the effective memory bandwidth for each kernel by dividing the total memory moved (in bytes) by its runtime. On most systems a double precision floating point number is 8 bytes, so for the Triad kernel $24N$ bytes are moved during its execution.

Table 3.1 also shows the balance of load and store operations in the kernels. Both Copy and Multiply read and write an equal amount of data, whilst Add and Triad loads twice as much as it stores. Dot on the other hand only reads data and does not write to main memory at all. Therefore these kernels may highlight asymmetry in an architecture's memory system. The computational intensity according to the cache-aware Roofline model of each of these kernels is low and therefore they fall into the memory bandwidth bound section of the 'roof'.

3.2.1 Other memory bandwidth benchmarks

The STREAM benchmark is not the only attempt to measure memory bandwidth. A number of benchmark suites include metrics to measure this, and their limitations are discussed in this section. Additionally many device vendors produce their own benchmarks which measure memory bandwidth. In particular the CUDA samples from NVIDIA contain a memory bandwidth example, however this uses a memory copy Application Programming Interface (API) call which does not contain any FLOPs rather than execution of a STREAM-esque kernel.

The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite is a large collection of small and large computational kernels used for testing the performance of heterogeneous systems [21]. The Triad kernel is included in this suite as the *deviceMemory* benchmark, however the time to transfer the arrays to and from the device are included in the achieved memory bandwidth calculation. This therefore does not capture the performance of the kernel itself with respect to the movement of memory between the execution units and device memory; it is this characteristic that the original STREAM benchmark captures and the upfront cost of memory allocation and initialisation is not included. Additionally, the movement of data arrays to a device are typically considered a one off initialisation cost and in an optimised application the data would be resident on the device as much as possible. Therefore the assumed pattern in this benchmark of the data remaining resident on the host is unrepresentative, and so is not comparable to STREAM in its measurement of memory bandwidth.

An OpenCL benchmark designed to measure memory bandwidth, *clpeak*, uses the vector types to implement a reduction [16]. The use of vector types in this way may result in non-contiguous memory access patterns. The OpenCL API supports querying the device for the preferred vector width of kernel implementations; a value which is typically one for modern GPUs and as such using the vector types inside the kernel is not the recommended approach any more.

The Standard Parallel Evaluation Corporation (SPEC) ACCEL benchmark suite contains a number of memory bandwidth bound kernels derived from real-world use cases [88]. The benchmark however does not include any of the STREAM kernels.

In summary therefore, beyond the STREAM benchmark itself, there are no other benchmarks which aim to measure the memory bandwidth under the same ethos; namely what memory bandwidth is achievable for representative computational kernels. BabelStream aims to fill this gap in a portable way.

3.3 The first BabelStream benchmark

The BabelStream benchmark is a new implementation of the STREAM kernels focusing on performance portability and platform portability. The original STREAM benchmark is written in C and uses the OpenMP programming model. As such it is only able to run on CPU style architectures and therefore unable to run on many-core devices, in particular GPUs. BabelStream was initially therefore first implemented in GPU suitable programming models, which also ran on CPUs too, in order to have a common benchmark for comparing the relative memory bandwidths between the multi-core and many-core devices. Much of the advantage of porting an application to run on a GPU was to take advantage of the improved memory bandwidth available there. BabelStream was then later extended to consider more programming models, as discussed in Section 3.4.

Whilst the STREAM benchmark has been a stalwart of CPU performance benchmarking, it is not possible to present results from other architectures as the benchmark simply does not run there. BabelStream therefore allows results to be generated across different architectures including CPUs, and so represents a fair test of achievable memory bandwidth. Because the kernels are also identical to STREAM, we should expect the numbers to be comparable. In a similar way to LINPACK providing a cross-platform, single unit of comparison, the BabelStream benchmark also serves a similar function; the same code may be run on many different systems in a fair way so that they may be directly compared.

The allocation, initialisation and the transfer of memory to the GPUs are not included in our timing measurements. The execution time of the kernel itself is used in the memory bandwidth calculation. This allows us to calculate the bandwidth of memory for the computational kernel alone. As such a synchronisation is required at the end of each kernel in order to generate accurate timings. The kernels therefore must behave as blocking calls with respect to the host. In self-hosting situations such as on CPUs this results in a synchronisation between threads, whereas on attached devices such as GPUs this results in a synchronisation between the device and the host. In both cases this ensures that from the perspective of the kernel driver (containing the timing routines) the computation has completed. The original STREAM benchmark also synchronises after each kernel and does not begin the next kernel without synchronising.

3.3.1 Initial results

The first version of BabelStream was implemented in OpenCL and CUDA, and consisted of the original four STREAM kernels. It was originally named GPU-STREAM as its primary focus was a GPU capability for measuring memory bandwidth. This allowed GPU results to be generated across NVIDIA and AMD consumer and HPC GPUs. Additionally the Intel Xeon Phi (Knights Corner) (KNC) co-processor and Intel CPUs are able to run both STREAM and the OpenCL version of BabelStream and so direct comparisons can be made.

These initial results shown in Figure 3.4 allow a single graph such as this to be produced for the first time, displaying the achieved memory bandwidth along with the percentage of theoretical peak taken from the device technical specifications. The achieved bandwidth (lighter colour) is overlaid on to the theoretically achievable peak (darker colour), with the percentage of achieved peak printed as labels. Of the devices tested at the time, the AMD Fury X GPU achieves the highest memory bandwidth, thanks to its use of HBM. The NVIDIA K40 offers the highest memory bandwidth of NVIDIA's HPC focused line; note that only one of the K80's two GPUs was used in this experiment. The AMD S9150 HPC GPU achieves a higher memory bandwidth than all of NVIDIA HPC devices, and is close in performance to the NVIDIA consumer GPUs. Such claims as these are made possible by having a single graphic which measures the achievable memory bandwidth in a fair way.

What is clear to see in this figure is that the OpenCL implementation on Intel's CPU architectures is somewhat lacking, and this is improved in subsequent versions of the benchmark. This lack of performance is primarily down to the copying of OpenCL buffers between the host and device not being non-uniform memory access (NUMA) aware. The performance measured for the McCalpin STREAM on KNC is only 137 GB/s, representing 43% of theoretical peak memory bandwidth, and so the OpenCL performance is similar on this device; however neither are particularly high.

3.3.2 The effect of error correcting code memory

With this benchmark the effect of error correcting code (ECC) memory on memory bandwidth on GPU architectures is also quantified. The GPUs that support ECC allow it to be turned on and off, and so the benchmark was run in both these configurations. Every byte in memory is checked for errors with a single bit; alternatively every 8 bytes is checked with a single byte. As such an extra byte must be read for every eight bytes read. This also has the side effect that the memory capacity of the device is also reduced by this ratio. The GPUs in this study do not provide any extra hardware resource within the memory controller for the extra bytes. As such achievable memory bandwidth is lost, resulting in 12.5% of the theoretical peak memory bandwidth being lost.

This behaviour is also replicated for the achieved memory bandwidth as shown in Figure 3.5 (lighter colours in the figure). The BabelStream benchmark

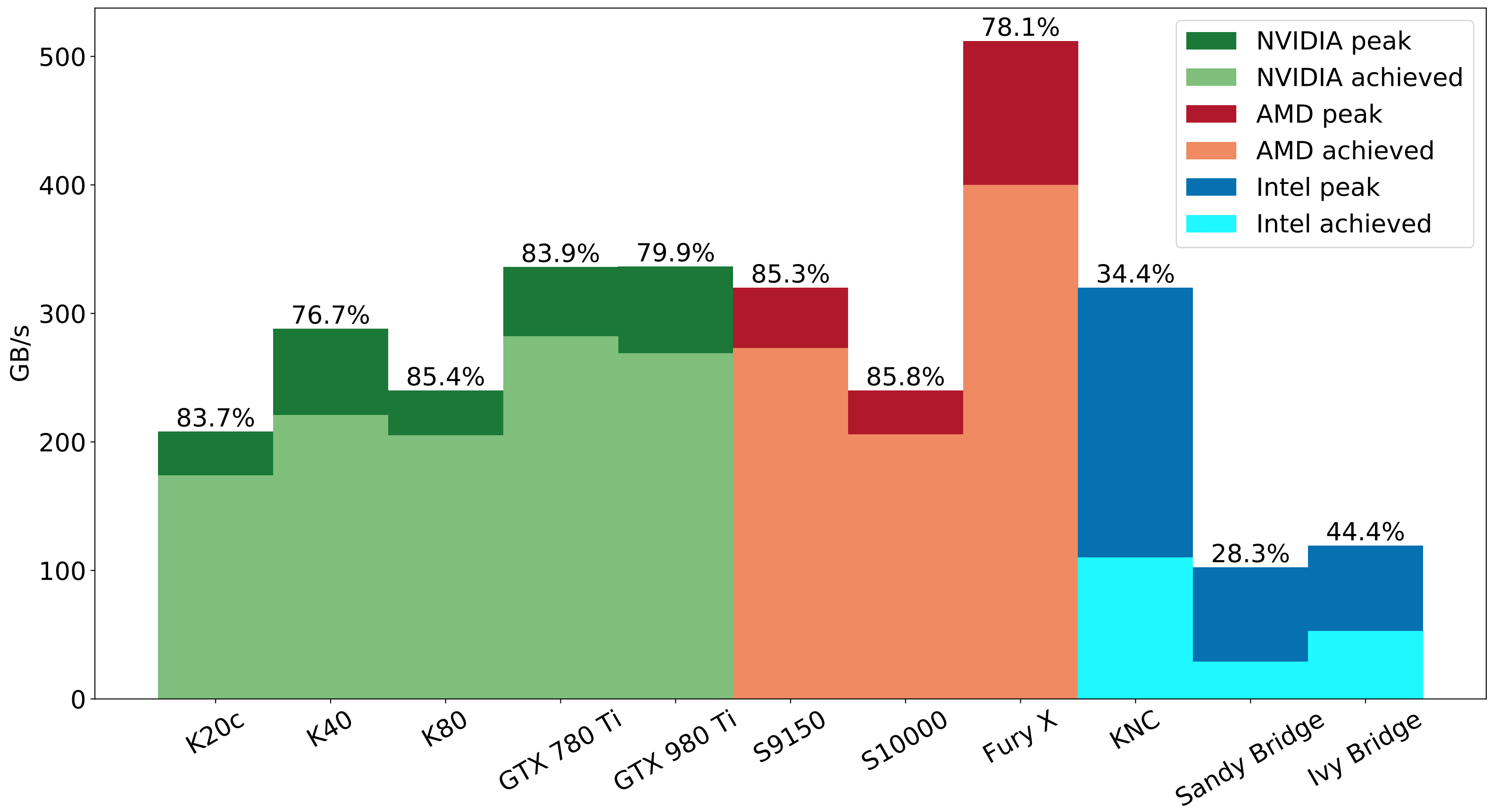


Figure 3.4: BabelStream v1 Triad memory bandwidth across devices (from [23])

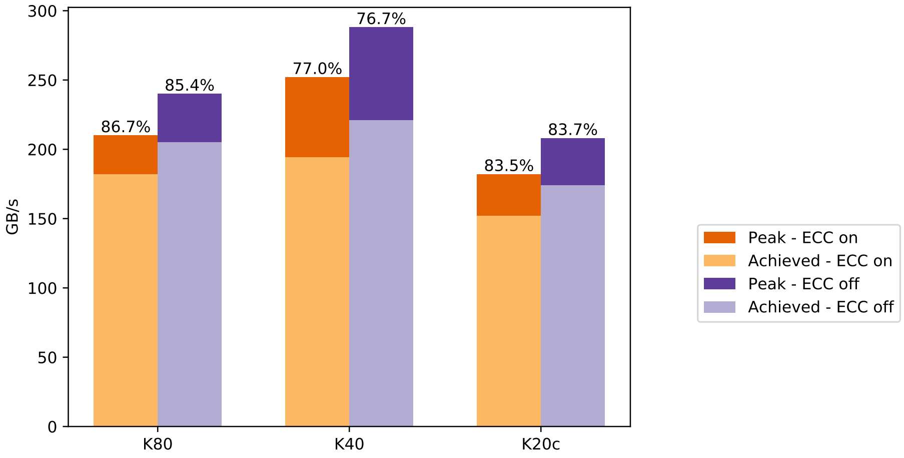


Figure 3.5: Effect of ECC on achievable memory bandwidth on NVIDIA HPC GPUs (from [23])

achieves similar percentages of peak memory bandwidth independent of the ECC settings, and as such the effect of performing the ECC parity check is not significantly impacting what the achievable bandwidth is. Rather the cost of reading the extra memory reduces the *available* memory bandwidth for the application itself. On CPUs it is not possible to turn ECC memory off in this manner as it is determined via the memory controller. However, CPU memory controllers use extra hardware resource, extra wires, to read the extra byte, and so memory bandwidth would not be negatively effected by this.

3.4 Expanding BabelStream with multiple programming models

When selecting a parallel programming model in which to write a high performance code, one would hope that the choice does not limit the ability to reach the respective hardware limits of the device. Many of the programming models also purport to allow a degree of performance portability, and at least functional portability. As such the BabelStream benchmark was rewritten to survey the current terrain of what memory bandwidth can be reasonably achieved for the STREAM kernels, across a wide range of devices and programming models. The simple kernels are written according to community best practices employed in scientific codes; for example using signed loop iterator variables (to mitigate index integer overflow concerns) and using `restrict` on pointers to prevent aliasing. Ultimately, if simple STREAM kernels are unable to achieve close to theoretical peak memory bandwidth limits, it is unrealistic to expect a large, complex, scientific code with memory bandwidth bound kernels to also achieve these limits.

As a side effect, this Open Source code can provide the HPC community with a sort of *Rosetta Stone* for each of the programming models. Firstly as a translation resource for understanding new programming models; as the kernels implemented are the same, the code acts as a way to compare the syntax required to write codes in a particular model. Secondly, it can serve as a place for demonstrating such best practices for each of the programming models in order to obtain an acceptable degree of performance portability.

The programming models used represent a range of abstractions and maturity. OpenCL and CUDA are relatively low level abstractions. OpenMP and OpenACC are compiler directive based approaches which allow existing codes to be annotated. Kokkos, RAJA and SYCL are all C++ abstraction layers describing the computational kernel as a lambda function executed under a parallel policy, a high level abstraction. Each of these models are sufficient to describe the parallelism for the STREAM kernels and there is nothing in any of their memory models which would prevent them from achieving peak memory bandwidth. As such this work represents a survey of the current capability of the implementation of the models on a wide variety of hardware.

The initialisation of data on the device is done with an initialisation kernel instead of a simple memory copy. This is to ensure that any NUMA effects are mitigated, most often caused from the first-touch policy of memory allocation: the memory is allocated in the NUMA region where it is first used by a core in that region. On GPUs this makes no difference, even for those devices with multiple memory controllers such as the NVIDIA P100: the bandwidth differed by an insignificant 0.01% between the CUDA implementation of BabelStream using a memory copy API or a initialisation kernel; it is likely that the latency tolerance of GPUs masks any measurable differences here. Note that the memory bandwidth of the CPU to GPU interconnect (such as PCIe) has much lower bandwidth than GPU main memory bandwidth and so therefore in an application with a large memory footprint it may be best practice to initialise the data on the device if possible for best performance. However on CPUs this is important; a memory copy would initialise all the data in one NUMA region and therefore a penalty would apply for access to this memory from other NUMA regions. As such the implementation of the kernels in each model is allowed to allocate and initialise the memory in a suitable manner; an approach consistent with a large application utilising a single programming model.

3.5 BabelStream performance

The BabelStream benchmark was run on 14 different devices from 4 hardware vendors: NVIDIA, AMD, Intel and IBM. The range of hardware covers both HPC and consumer GPUs, CPUs and the KNL processor, and is listed in Table 3.2. All the CPU systems are dual socket and hence the theoretical peak bandwidth is multiplied by two as shown. The theoretical peak memory bandwidth figure for KNL is not available and so the quoted value is based on the claimed five times DDR bandwidth. The KNL was booted into Flat/Quadrant mode and memory was allocated directly into MCDRAM via the `numactl` tool.

These devices were unfortunately not available in a single system and so a variety of platforms were used. The University of Bristol experimental cluster, named the ‘Zoo’ housed the consumer NVIDIA GPUs (GTX 980 Ti and Titan X), all of the AMD GPUs and the KNL. The K20X, Haswell and Broadwell CPUs were situated in the Cray XC40 supercomputer ‘Swan’ and the K40 and K80 were in the Cray CS cluster ‘Falcon’. The P100 was available in the

Cray XC50/XC40 supercomputer 'Piz Daint' at Swiss National Supercomputing Centre (CSCS). The Power 8 CPUs were made available from an Advanced Systems Technology Test Bed at Sandia National Laboratory (SNL). The Sandy Bridge CPUs were from BlueCrystal Phase 3 from the Advanced Computing Resource Centre at the University of Bristol. The Ivy Bridge CPUs were part of the Cray XC30 supercomputer 'Edison' at National Energy Research Scientific Computing Center (NERSC).

Due to the plethora of devices, programming models and systems used to gain the large coverage of combinations used for experimentation, a wide variety of compilers was used in order to build the benchmark for each combination of device and model. The exact combinations are shown in Table 3.3 and Table 3.4. For the SYCL results on NVIDIA GPUs, the ComputeCpp compiler generates SPIR, which the NVIDIA OpenCL driver did not support; therefore an experimental NVIDIA back end to pocl[1] was used for these cases. Whilst it may have been preferable to use the PGI compiler for all OpenACC results, it was unavailable on some of the systems. Note too that the highest x86 CPU architecture supported by the available version of the PGI compiler was Haswell, and so this was set as the target architecture for the Broadwell and KNL processors.

3.5.1 Triad performance

Implementations of BabelStream benchmark in each of the programming models were run on each of the devices listed previously. The arrays were of length 2^{25} double precision elements, which means each of the arrays were 256 MiB and as such are larger than any of the caches on the devices; therefore the data must be read from main memory. The scalar data type was used in all cases (rather than vector data types such as `double4`) as they are supported by all the tested models (unlike the vector data types) and as discussed in Section 3.2.1 modern devices prefer this approach. 100 iterations were run with the minimum runtime used to calculate the memory bandwidth, excluding the first iteration. Some implementations of the programming models themselves prevent running the benchmark on some devices, and so results cannot be presented in these cases. This is down to the implementation of the model rather than a fundamental aspect of it which is incompatible with some devices. The results for the Triad kernel are representative of the other STREAM kernels and so these results will be discussed, and the observations and conclusions apply to the other kernels. The Dot kernel will be discussed separately in Section 3.5.2. As before, the percentage of peak memory bandwidth will be calculated for the kernel.

The results in this section will be presented using a heatmap so that a value for each programming model executed on each device can be seen; darker colours represent higher (better) values. Where no result is possible, primarily due to unsupported combinations of programming model and device, it will be marked as 'N/A'. As an example, the original STREAM benchmark (labelled McCalpin) is not able to run on GPUs and as such these results are marked 'N/A'. There are a surprisingly large number of unsupported combinations, however this is down to vendor support rather than due to an incompatibility or issue with the programming model. As such the missing results should be seen as a weakness of the ecosystem surrounding each programming model. Note however that

[1] `http://portablecl.org`

Name	Architecture	Class	Vendor	Memory type	Memory BW (GB/s)
K20X	Kepler	GPU	NVIDIA	GDDR5	250
K40	Kepler	GPU	NVIDIA	GDDR5	288
K80 (1 GPU)	Kepler	GPU	NVIDIA	GDDR5	240
GTX 980 Ti	Maxwell	GPU	NVIDIA	GDDR5	224
Titan X	Pascal	GPU	NVIDIA	GDDR5X	480
P100	Pascal	GPU	NVIDIA	HBM2	732
S9150	Hawaii	GPU	AMD	GDDR5	320
Fury X	Fiji	GPU	AMD	HBM	512
E5-2670	Sandy Bridge	CPU	Intel	DDR3	2×51.2=102.4
E5-2697 v2	Ivy Bridge	CPU	Intel	DDR3	2×59.7=119.4
E5-2698 v3	Haswell	CPU	Intel	DDR4	2×68=136
E5-2699 v4	Broadwell	CPU	Intel	DDR4	2×76.8=153.6
Xeon Phi 7210	Knights Landing	MIC	Intel	MCDRAM	∼5×102 = 510
Power 8 @ 3.69 GHz, 8 core	POWER	CPU	IBM	DDR4	2×192=384

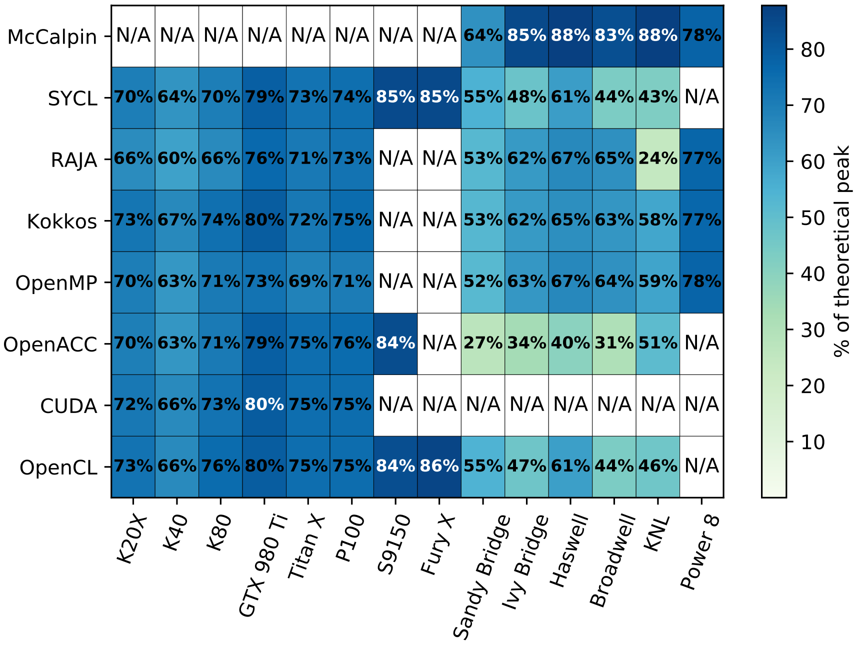
Table 3.2: List of devices used in BabelStream experiments (from [28])

Model	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X
GPU Driver	352.68	361.93.02	361.93.02	370.28	370.28	375.20	1912.5	1912.5
RAJA	GNU 5.3	GNU 4.9	GNU 4.9	GNU 4.9	GNU 4.9	GNU 5.3	N/A	N/A
Kokkos	GNU 5.3	GNU 4.9	GNU 4.9	GNU 4.9	GNU 4.9	GNU 5.3	N/A	N/A
OpenMP	CCE 8.5.5	CCE 8.5.5	CCE 8.5.5	clang-ykt	clang-ykt	CCE 8.5.5	N/A	N/A
OpenACC	PGI 16.10	CCE 8.5.5	CCE 8.5.5	PGI 16.7	PGI 16.7	PGI 16.9	PGI 16.7	N/A
CUDA	7.5	7.5	7.5	7.5	8.0	8.0	N/A	N/A
OpenCL	-	-	-	-	-	-	-	-
SYCL	ComputeCpp CE0.1.2							

Table 3.3: Compiler configurations for BabelStream experiments on GPUs (from [28])

Model	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Knights Landing	Power 8
McCalpin	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
RAJA	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
Kokkos	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
OpenMP	Intel 16.0	Intel 16.0	Intel 16.0	Intel 16.0	Intel 17.0	XL 13.1.4
OpenACC	PGI 16.10	PGI 16.10	PGI 16.10	PGI 16.10	PGI 16.10	XL 13.1.4
OpenCL	Intel 16.1	Intel 15.1	Intel 15.1	Intel 15.1	Intel 16.1.1	N/A
SYCL	ComputeCpp CE0.1.2					N/A

Table 3.4: Compiler configurations for BabelStream experiments on CPUs (from [28])



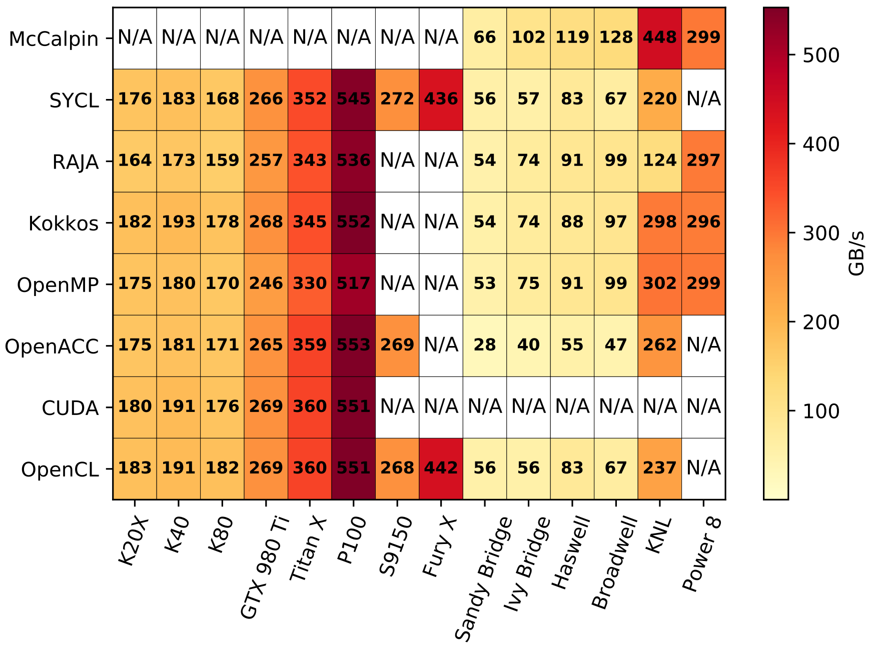
	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	64%	85%	88%	83%	88%	78%
SYCL	70%	64%	70%	79%	73%	74%	85%	85%	55%	48%	61%	44%	43%	N/A
RAJA	66%	60%	66%	76%	71%	73%	N/A	N/A	53%	62%	67%	65%	24%	77%
Kokkos	73%	67%	74%	80%	72%	75%	N/A	N/A	53%	62%	65%	63%	58%	77%
OpenMP	70%	63%	71%	73%	69%	71%	N/A	N/A	52%	63%	67%	64%	59%	78%
OpenACC	70%	63%	71%	79%	75%	76%	84%	N/A	27%	34%	40%	31%	51%	N/A
CUDA	72%	66%	73%	80%	75%	75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCL	73%	66%	76%	80%	75%	75%	84%	86%	55%	47%	61%	44%	46%	N/A

Figure 3.6: Fraction of theoretical peak memory bandwidth obtained by the BabelStream Triad kernel (from [28])

any missing results does mean a lack of portability, until an implementation is released. Support for NVIDIA GPUs is good and so these results are complete. AMD GPUs however suffer as at the time of writing it is not possible to directly compile and run CUDA, OpenMP, Kokkos or RAJA codes. The GPU support for both Kokkos and RAJA has been implemented using CUDA. Support for OpenACC on these devices has also been dropped, however older versions of the compiler still support some of the other GPUs. OpenACC support in the GNU compiler is only for APUs and not the discrete GPUs tested here. PGI has also dropped CUDA-x86 support, and so there are no results for CUDA on Intel devices; with old versions of this compiler significant source changes were required in order to run the benchmark. At present there are few programming models supported on the IBM Power 8 — only OpenMP, and therefore Kokkos and RAJA with their OpenMP back end.

The achieved percentage of peak memory bandwidth for the Triad kernel is shown in Figure 3.6 with the raw memory bandwidth achieved shown in Figure 3.7. Results will be discussed by vendor as many of the comments apply to multiple of their devices; see Table 3.2 for the list of devices by each vendor.

NVIDIA GPUs The P100 achieves the highest memory bandwidth of all the devices tested due to its use of HBM2 technology. The achievable performance is consistent both across generations of GPU and the choice of programming model with all results within 60–80% of theoretical peak memory bandwidth. Note too that CUDA is considered to give a baseline performance and each of



	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	66	102	119	128	448	299
SYCL	176	183	168	266	352	545	272	436	56	57	83	67	220	N/A
RAJA	164	173	159	257	343	536	N/A	N/A	54	74	91	99	124	297
Kokkos	182	193	178	268	345	552	N/A	N/A	54	74	88	97	298	296
OpenMP	175	180	170	246	330	517	N/A	N/A	53	75	91	99	302	299
OpenACC	175	181	171	265	359	553	269	N/A	28	40	55	47	262	N/A
CUDA	180	191	176	269	360	551	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCL	183	191	182	269	360	551	268	442	56	56	83	67	237	N/A

Figure 3.7: Sustained memory bandwidth achieved by the BabelStream Triad kernel (from [28])

the other models does achieve close to the CUDA performance, indicating that at least for this vendor all the programming models provide a way to exploit the memory bandwidth of these devices. In some cases the CUDA performance is exceeded; this will be down to CUDA requiring the programmer to make a choice of the thread block size, which for BabelStream is set to 1024 [87]. This number was chosen empirically to provide good performance across the different architectures. The necessity for the programmer to specify a value such as this may reduce the performance portability of a such a code. Selection of a value limit the representativeness of the Triad kernel to an application, however it is simple to adjust this parameter. Minor tuning of this parameter for specific incarnations of the architecture may also be required, and is often recommended. The models which do not require the programmer to set this value may make such informed choices per device mitigating this issue.

AMD GPUs The main conclusion that can be drawn from the results of BabelStream is that there is currently a widespread lack of support for AMD GPUs. Therefore the only viable choice to the programmer is the low level OpenCL, which for large scientific codes is unpalatable. It is hoped that in time support for the other higher-level models will grow. Despite this however, the SYCL abstraction over OpenCL shows little overhead above OpenCL directly. For the data points that were able to be collected, they all achieved 84–86% of the theoretical peak memory bandwidth which is the highest fraction of all the devices tested.

Intel processors The McCalpin STREAM results are considered the gold standard metric for memory bandwidth and so it is these results that are used as a baseline for the BabelStream results on Intel processors. Where possible, one thread was launched per physical core and this was set via specific environment variables for each of the models. The theoretical peak memory bandwidths from generations of Intel CPUs has improved as shown in Table 3.2 and these also manifest into the achievable bandwidth as shown in Figure 3.7; from Sandy Bridge to Broadwell the achievable bandwidth has almost doubled from 66 GB/s to 128 GB/s.

However many of the programming models do not achieve close to the McCalpin value without some additional care [83]. The McCalpin STREAM benchmark sets the size of the arrays at compile time, and so the compiler is therefore able to use this knowledge to inform the optimisation passes. Specifically, the memory is allocated on the stack and so the compiler can ensure this is aligned, and therefore issue aligned load and store instructions for the memory accesses. Additionally, the reuse heuristics can be accurate as the trip count of the loops are known, and combined with the knowledge of alignment, non-temporal stores can be issued so that pollution of the cache with the output array of each kernel does not occur. It is unrealistic for a real scientific code to embed the problem sizes in this way, and therefore knowledge of the problem size is generally only available at runtime. In this way STREAM in unrepresentative of today's production codes.

The McCalpin STREAM benchmark can be modified accordingly and the achieved memory bandwidth is indeed reduced; on KNL it is reduced to 270–345 GB/s (there is now a large variance between runs due to unaligned memory). This bandwidth is in line with the out-of-the-box performance of BabelStream. On examining the compiler optimisation reports it can be found that unaligned memory access instructions are issued and non-temporal stores are not issued.

Therefore some simple updates to the BabelStream benchmark were made by Karthik et al. to improve the performance [83]. These changes are simple enough that the lessons learnt should be considered best practice in line with the ethos of the benchmark. With OpenMP and RAJA, the memory allocations are aligned to 2 MiB pages using the C11 `aligned_alloc` call. With OpenMP the `simd aligned` clause was used to state that the addresses were aligned; a required step for generating aligned memory accesses due to the location of the data on the heap. For OpenMP, RAJA and Kokkos, streaming stores were generated using the necessary Intel compiler flag; note that a compiler directive recognised by the Intel compiler could be used instead and it is this latter approach that would be recommended in a real application as it allows for more fine grained control of the generation of non-temporal stores. Extra steps were required to build the Kokkos and RAJA libraries to enable memory alignment and to prevent pointer aliasing to ensure vectorisation. Also due to type mismatches in the Kokkos and RAJA libraries the use of the `long` data type was required for the loop index variable instead of the usual `int`; this ensured that typecasts did not occur.

Updated Intel results with these changes are shown in Figure 3.8 which shows the improvement in comparison to Figure 3.7. Note now how the results are much closer to the original McCalpin STREAM results, and these relatively simple changes were required for this to be reached, but this does highlight the important point that programming languages and models only go so far into

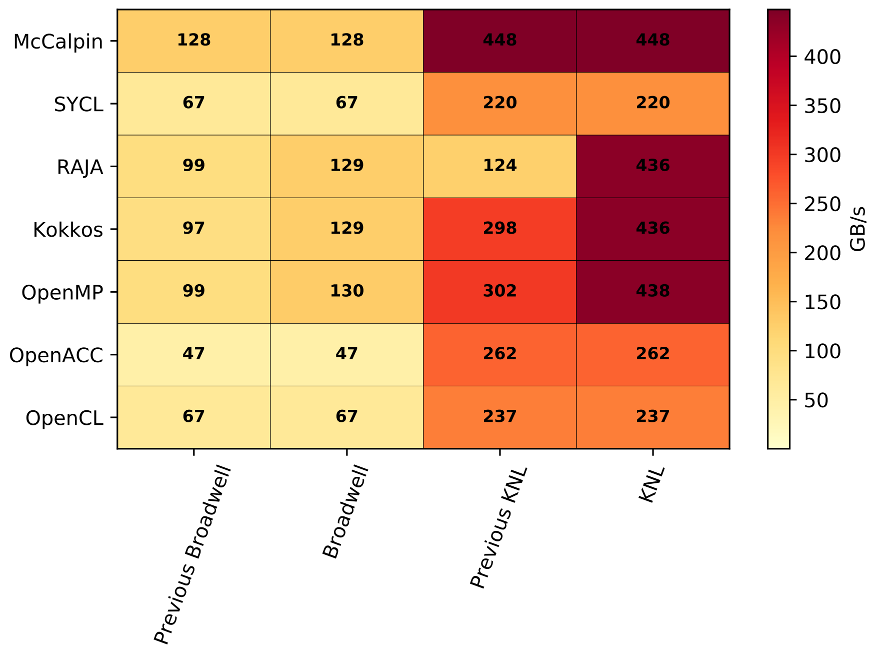


Figure 3.8: Sustained memory bandwidth achieved by the updated BabelStream Triad kernel on Intel architectures

describing what the programmer is trying to express.

The C++ abstraction layers Kokkos and RAJA again show good performance and little overhead over the performance of the model used in their back end (in this case OpenMP). However, the OpenACC and OpenCL performance is somewhat lacking. With OpenACC this is likely attributed to NUMA effects, as although the initialisation of the data is done as a kernel rather than a memory copy so as to conform to the first-touch policy, the implementation of OpenACC itself on the specific system still seems to first-touch the arrays internally. Although use of the OpenACC API calls to allocate the memory directly improves the CPU performance (and therefore strengthens the observation that the issues found are due to NUMA), these do not function correctly on GPUs without further source changes.

The MCDRAM present on KNL shows a clear advantage over traditional DDR memory technologies on CPUs, and also the GDDR of most of the GPUs tested including the AMD Fury X which does utilise HBM. However the HBM2 in the NVIDIA P100 still offers more memory bandwidth.

Power 8 Running STREAM on the Power 8 was shown by Reguly et al. to be sensitive to problem size [86], and with a much larger array size than used with BabelStream a higher bandwidth can be recorded; however for consistency the same problem size was used on all devices.

Compared to Intel CPUs (excluding KNL which uses MCDRAM), much higher bandwidth can be obtained from DDR as a result of the increased num-

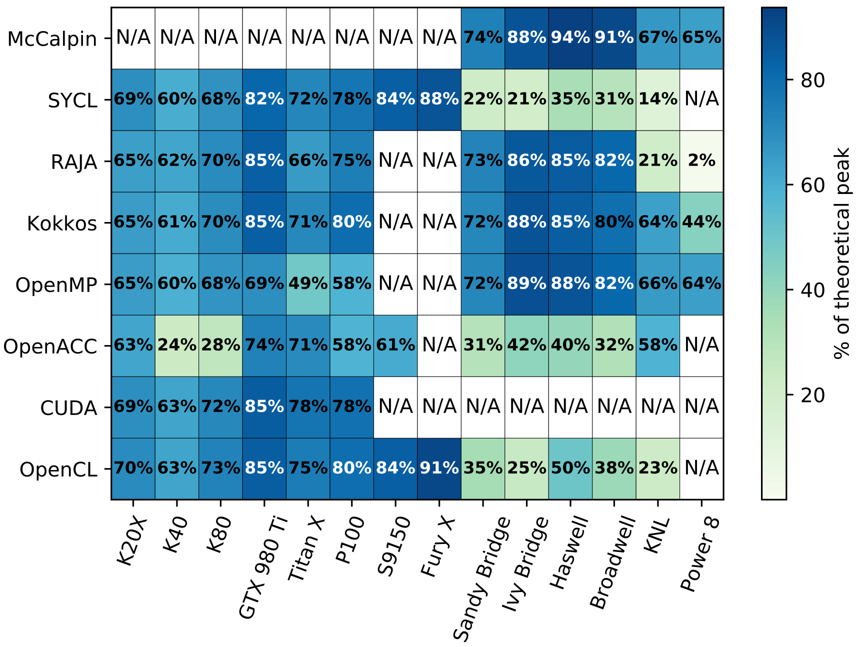


Figure 3.9: Fraction of theoretical peak memory bandwidth obtained by the BabelStream Dot kernel (from [28])

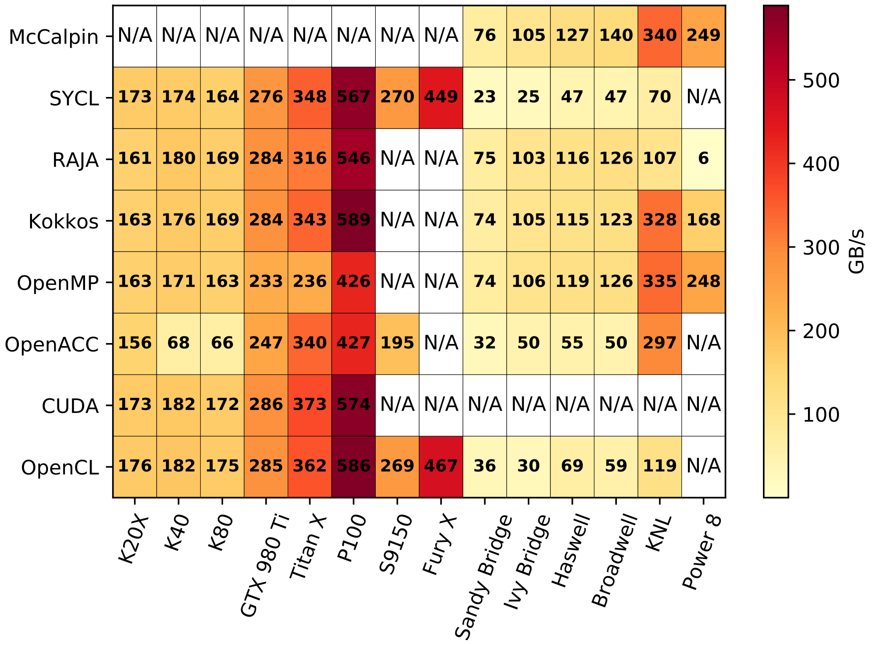
ber of memory channels and Centaur chips. Note that while extra steps were required for the BabelStream benchmark to match the McCalpin STREAM performance on Intel architectures, this was not required here. This is because the Power ISA v2.07 (as used for Power 8) does not have non-temporal store instructions, and as such the optimisations applied for Intel processors to generate such instructions do not help.

3.5.2 Reduction performance

For the Dot kernel BabelStream requires the result is made available on the host, and so the transfer of this single value result is included in the timing. The other kernels transform data and so transfer to or from the host is not necessary as such kernels in real scientific applications would not transfer these arrays before and after each kernel execution. On the other hand, the result of a reduction is often used on the host, for example in convergence checking. Additionally, requiring this value on the host ensures that the reduction completes.

The results are presented in a similar way to the Triad kernel, with the percentage of theoretical peak memory bandwidth shown in Figure 3.9 with the raw memory bandwidth achieved shown in Figure 3.10.

NVIDIA GPUs The NVIDIA Kepler architecture shows slightly reduced Dot performance compared to Triad, whilst the Pascal architecture shows increased performance over Triad, however these differences are slight at around 5%. The Cray compiler was used for the OpenACC results on the K40 and



	K20X	K40	K80	GTX 980 Ti	Titan X	P100	S9150	Fury X	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	KNL	Power 8
McCalpin	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	76	105	127	140	340	249
SYCL	173	174	164	276	348	567	270	449	23	25	47	47	70	N/A
RAJA	161	180	169	284	316	546	N/A	N/A	75	103	116	126	107	6
Kokkos	163	176	169	284	343	589	N/A	N/A	74	105	115	123	328	168
OpenMP	163	171	163	233	236	426	N/A	N/A	74	106	119	126	335	248
OpenACC	156	68	66	247	340	427	195	N/A	32	50	55	50	297	N/A
CUDA	173	182	172	286	373	574	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCL	176	182	175	285	362	586	269	467	36	30	69	59	119	N/A

Figure 3.10: Sustained memory bandwidth achieved by the BabelStream Dot kernel (from [28])

K80, and whilst Triad performance was in line with the other results, there is a performance issue with the reduction kernels on NVIDIA GPUs. This is also somewhat present with the open-source LLVM OpenMP compiler targeting these devices but is not present in the Cray compiler.

AMD GPUs OpenCL and SYCL required hand writing the reduction kernel, and so this performs well. OpenACC automatically implements the reductions, not the programmer, and Figure 3.9 shows that it does not perform well in comparison, however it is in line with the percentage of theoretical peak of the NVIDIA GPUs when using the PGI compiler.

Intel processors Unlike the Triad kernel, the Dot kernel initially performs close to McCalpin STREAM; this is because only read memory operations are required as there is no output array to pollute the cache. The OpenCL (and SYCL) kernel is perhaps designed with GPUs in mind. With a more cache-friendly memory access pattern the bandwidth can be improved, however this reduces the performance on the GPUs. Therefore in a production application utilising OpenCL where reductions are a large bottleneck, per-architecture reduction implementations should be written. The BabelStream benchmark should be single source; it does not re-specialise for particular architectures. This highlights an issue in the performance portability of reduction kernels across architectures and is a result of the low-level nature of OpenCL; higher level programming models can hide the complexity of device specific reduction operations, where as Figure 3.9 shows OpenMP reductions do similarly well across all architectures. As with Triad, OpenACC performance using the PGI compiler of the Dot kernel on CPUs is also poor.

The KNL processor does highlight an interesting point, where the Dot kernel achieves a significantly lower 340 GB/s compared to Triad at 448 GB/s. Whilst both Triad and Dot have two read operations, Triad writes to memory whereas Dot does not. It has previously been observed that as MCDRAM has separate read and write memory channels, streaming writes can be performed concurrently with reads, and so it is not possible to obtain the maximum memory bandwidth with read operations alone [49].

Power 8 In general the performance of the Dot kernel is more variable than the Triad kernel, and as with the KNL again achieves lower performance than Triad, around 20 GB/s less. The 192 GB/s of Power 8 memory bandwidth available to each socket is split into 128 GB/s read and 64 GB/s write [86] and as there are no write operations in this kernel the theoretical peak bandwidth of this dual-socket system should instead be close to 256 GB/s; indeed McCalpin STREAM achieves close to this value. Only on this architecture does Kokkos show any performance overhead over code written directly in OpenMP. The RAJA performance is severely lacking indicating an issue with the reduction implementation in the RAJA library itself.

3.6 A survey of performance portability

The BabelStream results provide a survey of a simple kernel executed across a wide variety of devices from different vendors, and in a variety of programming models. The previous analysis focused primarily on the performance of each model on a particular (class of) device. However it is also possible to examine how each programming model fares across devices without source changes. The results of the Triad kernel from Figure 3.6 will be used in this section.

Pennycook et al. have used the results presented in this chapter (and elsewhere [27]) in development of a metric for measuring performance portability [81, 82]. The percentage of theoretical peak memory bandwidth of BabelStream shown in Figure 3.6 and Figure 3.9 would be inputs into the metric. This metric is the harmonic mean of the efficiency of performance across a (sub-)set of platforms. Any unsupported architectures result in the metric reporting zero portability, as functional portability is a minimum requirement. To this end, the authors restrict their analysis to a minimum supported set of platforms by all models (thus excluding IBM and AMD devices) so that non-zero numbers can be generated.

Both OpenCL and SYCL ran across almost all of the devices tested, but whilst they provided good performance on GPUs the CPU performance was lower. This behaviour is also seen in the performance portability metric of Pennycook et al. , which classified OpenCL with 46.2% application efficiency on CPUs and 99.7% on GPUs [82].

OpenMP also fared well across CPUs and GPUs, however the current lack of support for AMD GPUs reduces its portability at present. The offload and native compiler directives are different, and therefore different directives are required depending on which execution model is being used. Although this was a simple endeavour in BabelStream via preprocessor macros, in a larger code base this may become cumbersome. Again, the performance portability metric shows this, with an overall application efficiency of 82.1% [82]. This was calculated without the optimisations for Intel architectures discussed in 3.5.1, and so this metric would likely increase if recalculated with these updated results.

Although OpenACC only requires a single directive across all architectures, the lack of performance on all CPUs is worrisome along with the lack of support for all the vendors (at present OpenACC is driven mainly by PGI which is owned by NVIDIA). The performance portability metric reflects this too: whilst it achieves 95.6% on NVIDIA GPUs, on CPUs it is only 50.0% (with 63.5% taking into account both types of device) [82]. Some of the performance issues on CPUs may be down to the configuration of the specific systems chosen, however this is often outside the application developers control; for this thesis other platforms were not available with the appropriate software to verify this more robustly.

Although CUDA achieved among the best performance on NVIDIA GPUs, it is not portable to CPUs or to GPUs from different vendors and therefore cannot be considered performance portable. This is one such example where the performance portability metric is reported as zero by Pennycook et al. due to being unable to run on more than one class of processor [82].

Both RAJA and Kokkos show good performance portability between CPUs and GPUs, however it should be noted that RAJA does not include memory abstractions and so the programmer must perform this manually via another API (`malloc` for CPUs or CUDA for NVIDIA GPUs). Kokkos allowed a single source change to switch between CPUs and GPUs with no other changes required; the movement of memory between the devices is built into the abstraction. Neither model currently supports AMD GPUs. The performance portability metric (which excludes AMD GPUs in its calculation) shows that much like OpenMP, these models achieve around 85% of performance portability across both (Intel) CPU and (NVIDIA) GPU architectures [82].

3.7 Summary

BabelStream provides a comprehensive benchmark which can measure the achievable memory bandwidth on a range of devices in a range of programming models. As such it allows for a historical record to begin which tracks the performance of both programming models and hardware, and allows simple evaluations of the support offered by vendors for specific combinations of device and model. This record can be found online at `http://uob-hpc.github.io/BabelStream/`, which contains a version controlled repository of the code and the results. The results presented in this thesis were collected with `v2.0` of the benchmark, with the output of the benchmark also saved in the repository. This version of the code has also been assigned a DOI of `10.5281/zenodo.1203607`.

BabelStream has also allowed for further study into attainable memory bandwidths from additional levels of the cache hierarchy [29].

The benchmark shows that at the time of writing the choice of programming model is largely a matter of preference, where close to peak performance is achievable on a range of hardware in any particular model. Notable exceptions are where the model does not support a particular device at present. The performance portability metric of Pennycook et al. also quantified the degree of performance portability, drawing similar conclusions [81, 82]. As such the results generated by this benchmark provide the maximum achievable bandwidth which may be used to calculate the relative sustained memory bandwidth of other memory bandwidth bound kernels; such analysis will be used in later chapters.

The benchmark also allowed further exploration into the memory hierarchy of Intel processors; for example, ensuring memory alignment and non-temporal stores were generated for this benchmark which ensured the best performance. This highlighted issues with treating the McCalpin STREAM benchmark as representative of today's applications, which unlike STREAM have data allocated on the heap and the problem size unknown at compile time. As such the compiler receives less information, yet as programmers we expect a similar level of optimisation. The optimisations presented show how to leverage the most of the memory bandwidth.

CHAPTER 4

The computational nature of deterministic transport

The transport equation is a Boltzmann integral-differential equation which models the movement of neutral particles. In all but the simplest cases, the equation cannot be solved analytically, and so an approximate solution is sought using numerical methods. This chapter will first briefly introduce the transport equation itself and describe the constituent terms which model the various interactions of the particles with the material. The solution of this equation on a computer is explained. The proxy application used as the basis for this work is introduced and compared to alternative applications, and other non-transport applications which on the surface have some similarities with a transport solver.

4.1 The transport equation

$$\left[\frac{1}{v}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\vec{\nabla} + \sigma(\vec{r},E)\right]\psi(\vec{r},\hat{\Omega},E,t) = \tag{4.1a}$$

$$q_{\text{ex}}(\vec{r},\hat{\Omega},E,t) + \tag{4.1b}$$

$$\int dE' \int d\Omega' \sigma_{\text{s}}(\vec{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\psi(\vec{r},\hat{\Omega}',E',t) \tag{4.1c}$$

The deterministic transport equation (4.1) (from [55]) describes the interaction of neutral particles with materials. The notation used in this equation is listed in Table 4.1. The neutral particles may be neutrons or photons. The particles are considered to travel in straight lines and there are no particle-to-particle interactions (the particles react with the material and not each other). The interaction with the material is described in terms of a cross section, which describes the number of collisions each particle could make with the material per unit length travelled. The transport equation describes the net balance of the neutral particles in the material as they move through it. The particles may simply leave the material boundary, known as streaming, or may be lost to absorption.

Symbol	Description
$\hat{\Omega}$	Direction vector — in 3 dimensions this is (μ, η, ξ). Only two of these directions are required to define the vector uniquely as $\cos^2\mu + \cos^2\eta + \cos^2\xi = 1$
$\vec{\nabla}$	Directional derivative operator: $\hat{e}_x \frac{\partial}{\partial x} + \hat{e}_y \frac{\partial}{\partial y} + \hat{e}_z \frac{\partial}{\partial z}$
ψ	Angular flux
$\vec{r}$	Position vector (x, y, z)
E	Energy
q_{ex}	External source
σ_{s}	Scattering cross section
$\chi(E)$	Probability density function of fission neutron energy
$v(E)$	Mean number of neutrons at energy E
σ	Macroscopic cross section

Table 4.1: Transport equation notation

Cross sections, denoted σ, are usually made up of two parts which describe absorption and scattering of the particles respectively. In the case of neutrons, when the particle collides with a material nucleus, it may be absorbed and become part of the nucleus itself, or bounce off changing direction of travel and/or (kinetic) energy. The scattering cross section describes those particles which change direction and energy.

The cross sections in the equation are more formally known as macroscopic cross sections. They are derived from the microscopic cross sections, which describe the probability of an interaction with a (single) material nuclide. The total amount of material along with its density and the microscopic cross sections are used to generate the macroscopic cross sections. These are stored in a simple table for use during the transport solver and so the generation of this data is not important to this thesis.

The equation is constructed in terms of cross sections and the angular flux. The angular flux describes the expected number of particles in a volume travelling in a particular direction with a particular energy. The angular flux is denoted ψ. Seven dimensions are typically required to describe the angular flux: $\psi(\vec{r}, \hat{\Omega}, E, t)$ for 3D spatial position $\vec{r}$, 2D angular direction $\hat{\Omega}$ and energy E at time t.

The scalar flux is defined as the integration over the angular domain of the angular flux:

$$\phi(\vec{r}, E, t) = \int \psi(\vec{r}, \hat{\Omega}, E, t)\ d\Omega \tag{4.2}$$

The transport equation therefore describes the change in time of the particles. The loss of particles due to streaming is given by the $\hat{\Omega} \cdot \vec{\nabla}\psi(\vec{r}, \hat{\Omega}, E)$ term. The loss of particles due to collisions is given by the $\sigma(\vec{r}, E)\psi(\vec{r}, \hat{\Omega}, E)$ term. Collectively they are known as the streaming-collision operator. This completes the left hand side of (4.1a), along with the change in time.

The right hand side of the equation is often presented as a single term known as the source term $q(\vec{r}, \hat{\Omega}, E)$. This is mainly for convenience during the solution of the equation itself. The source term is made up of the sum of an external source, a scattering source, and in some cases a fission source. We do not consider the fission source in this thesis. The gain of particles from some known source is given by $q_{\mathrm{ex}}(\vec{r}, \hat{\Omega}, E)$. This term is provided from some measured data. Particles may be gained into the particular angle and energy considered through scattering; that is particles changing direction and/or energy. The scattering source term is constructed to count particles scattering *into* the energy, space and angle that the equation is defined in terms of. The material will determine the properties of scattering and these are defined in terms of the scattering cross section. The scattering source is given by $\int dE' \int d\Omega' \sigma_{\mathrm{s}}(\vec{r}, E' \to E, \hat{\Omega}' \cdot \hat{\Omega}) \psi(\vec{r}, \hat{\Omega}', E')$. The source term is the sum of the external and scattering source.

4.2 Discretisation of the transport equation

As with many balance equations, the transport equation is defined on continuous domains. It must be discretised, itself an approximation, in order to be solved computationally. This section describes the discretisation schemes employed in each problem dimension.

4.2.1 Spatial discretisation via finite difference

Given a general function over a continuous spatial domain $f(x)$ we wish to discretise the domain. We divide the spatial domain into a series of contiguous cells, with centres x_i. Each cell occupies the range $(x_{i-1/2}, x_{i+1/2})$.

As with many discretisation schemes, the finite difference (FD) approximation proceeds by integrating the transport equation over the cell. The cell centered value is used to approximate this integral for the collision and scattering terms as the cross section data is also assumed to be piece-wise constant across each cell; that is the value is constant (the same) across the cell and $\sigma(x)$ is the same for $x \in (x_{i-1/2}, x_{i+1/2})$. As such the integral over the cell domain for these terms is assumed to be equal to its value at the midpoint.

Evaluating the integral for the streaming operator introduces angular flux terms centred at the cell boundaries. We use the central FD equations to close this relation. The central FD equation is:

$$\psi(x_i) = \frac{\psi(x_{i-1/2}) + \psi(x_{i+1/2})}{2} \tag{4.3}$$

The equation states that the central cell value is equal to the average of the value on the boundaries. These FD equations along with the transport equation form a system of equations to solve.

This FD equation can be simply rearranged to specify the value at the boundary in terms of the central value and the other boundary value. For example:

$$\psi(x_{i+1/2}) = 2\psi(x_i) - \psi(x_{i-1/2}) \tag{4.4}$$

$$\psi(x_{i-1/2}) = 2\psi(x_i) - \psi(x_{i+1/2}) \tag{4.5}$$

The orientation of this rearrangement is determined by the angular direction cosine from the Discrete Ordinates (S_n) discretisation of the angular domain. One rearrangement (4.4) or (4.5) is then substituted into the transport equation so that the outgoing angular flux term for each cell is eliminated so that the equation is defined only in terms of cell centred and incoming angular fluxes.

More details may be found in Appendix A, and a full treatment of the FD spatial discretisation in multiple spatial dimensions is shown in Lewis and Miller [55].

4.2.2 Angular discretisation

A Discrete Ordinates (S_n) angular discretisation is typically used to discretise the angular dimension. This is formed by approximating the integral in (4.2) of the angular flux over the angular domain to form the scalar flux using a quadrature rule:

$$\phi \approx \sum_n w_n \psi_n \tag{4.6}$$

where the w_n are known as the quadrature weights which each correspond to a particular (discrete) angular direction in the domain.

We can therefore solve the transport equation for each of these angles, and use the quadrature to integrate the angular flux to the scalar flux. The quadrature set is defined by the list of angle directions and their associated weights. In general these form a simple look up table. The quadrature set is denoted S_n, where n is the 'order' of the set.

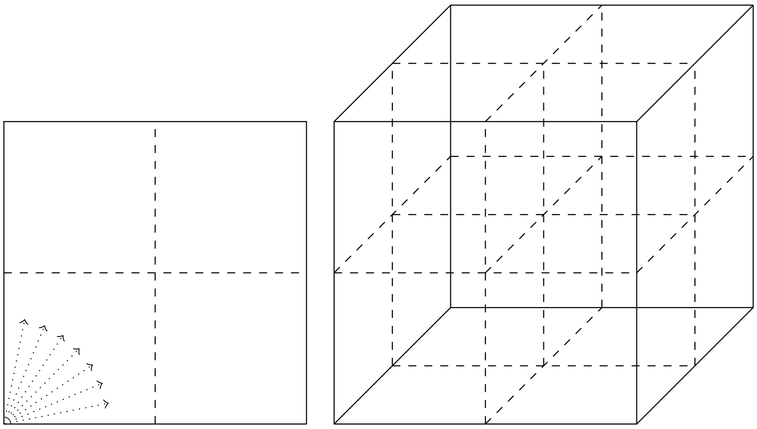
The angles can be grouped into quadrants (2D) and octants (3D) which describe their general direction with respect to the coordinate axis. In 3D, the angles can be defined in terms of their directional cosines (μ, η, ξ), the cosine of the angle between the direction and the reference axis. Angles with the same sign ± on each of these cosines belong to the same octant. Alternatively in a regular structured mesh, all angles which begin a sweep from the same corner cell are said to belong to the same octant. In this case the directed acyclic graph (DAG) which describes the sweep is identical for all angles in an octant (see Section 2.6 for a brief introduction to DAGs).

There are 8 octants in 3D and 4 quadrants in 2D, as shown in Figure 4.1. Angles for one of the quadrants are shown in Figure 4.1a. In this thesis, octants will be used to describe this grouping interchangeably. For a simple S_n quadrature with angles spread equally throughout the octants, there are $n(n+2)/8$ angles per octant.

4.2.3 Energy discretisation

The energy domain is discretised by dividing the domain into a number of energy groups. In doing so, limits are placed on the maximum and minimum energy which is modelled by the solution, and so these bounds are normally chosen to be suitably large to represent the significant portions of the domain. The energy is considered to be constant within the energy group. Any number of energy groups may be chosen, and they may be of differing sizes, although the size of each group has no impact on the nature of compute itself as only a single value is required per group. The number of groups does however contribute to the storage requirements of the scattering cross section and may affect the iteration count. An approximation is made such that the transport equation is considered to hold for each discrete energy group; and the resulting equation is known as the multi-group approximation.

The group-to-group scattering source term stipulates that contributions from the different groups are required to calculate the scattering source for a set



(a) Quadrants (b) Octants

Figure 4.1: Division of spatial mesh into quadrants and octants

group. Therefore the groups can be solved in descending order (in a Gauss-Seidel manner), or independently (in a Jacobi manner). Assuming the neutral particles are predominately scattered to lower energies (down-scattered), solving the groups under a Jacobi scheme means lower energy groups use the values from higher energy groups from a previous iteration, whereas the Gauss-Seidel scheme allows lower energy groups to use the values from higher groups for the current iteration. As such, a Gauss-Seidel regime should have improved convergence properties over Jacobi, and indeed this was shown to be the case with Jacobi requiring up to 10% more iterations [14, 13]. However, a Jacobi scheme does provide a source of additional concurrency. Both methods should however produce the same answer within a given tolerance once convergence has been reached.

4.3 Numerical solution

The solution to the transport equation is the angular flux ψ. It is also common to only require the scalar flux ϕ instead and so the solution may also be determined by this quantity. The scalar flux requires integration of the angular flux over the angular dimension and so the angular flux itself must be obtained first.

The transport equation is typically solved by simple iterations on the scattering source. The scattering source term (4.1c) depends on the angular flux; although it is integrated over angle and so the scalar flux may be used instead in the special case of isotropic scattering, where the outgoing direction is independent of the incoming direction. However it is precisely the flux that is the unknown in the equation for which the solution is sought. An initial estimate is made for the flux (often zero is used) so that the scattering source may be estimated, and as such the right hand side of the transport equation may be evaluated.

Given this estimate of the flux, the source terms (4.1b) and (4.1c) on the right hand side of the transport equation may be calculated and then the streaming-collision operator may be directly inverted to calculate the angular flux on the left hand side of the equation. Such a technique is known as an implicit solution method (solve) applied to the streaming-collision operator, and the reader is referred to a more thorough and general handling of solving linear systems with simple iterations and iterative methods by Greenbaum [37]. This gives an updated value for the flux which may then be used to recalculate the (scattering) source term on the right hand side, thus forming an iterative scheme which improves on the initial estimate of the flux and successively refines it. The process continues until the flux ceases to change significantly, and therefore the source terms would also cease to change with additional iterations and the iterative scheme ends.

If other sources are present on the right hand side then this method is applied to those terms also. For this thesis however, the source terms are as shown in (4.1) with a fixed external source and the scattering source only. Calculation of the sources themselves is in general simple as all points in the domain are independent, thus making a parallel implementation straightforward.

Importantly, it is during the inversion of the streaming-collision operator that a sweep across the spatial domain is required. The mathematics behind the dependency are well explained by Lewis and Miller [55], but in short there is a data dependency so that each cell requires boundary data from upwind neighbours, with the 'wind' direction determined by the S_n angle (recall Section 4.2.1). Therefore unlike many other solvers of (integro-)differential equations, in particular those in fields such as computational fluid dynamics, heat conduction and Lattice Boltzmann methods, each cell cannot be solved simultaneously.

Issues such as accuracy, the convergence properties of the algorithm and conserved quantities are relevant only to the physics aspects of the solution and hence little reference will be made to them. The accuracy of the solution depends on the properties of the discretisation used for each of the problem dimensions, and in general refers to the difference between the obtained solution and the 'true' solution. The convergence properties of this scheme will produce a small or large number of iterations required in order for the solution to be found. A choice in the energy domain of a Jacobi or Gauss-Seidel scheme as explained in Section 4.2.3 contributes to the iteration count. Lewis and Miller describe the iterations on the scattering source as having a physical interpretation, in that each iteration corresponds to a particle collision [55], and so the number of iterations is also dependent on the specific data used. Additional approximations may be made in order to accelerate the convergence, however these are not considered as part of this thesis which focuses on the computational nature of the sweep. The transport equation itself conserves the neutral particles (such as neutrons or photons), and the multi-group formulation of the transport equation ensures that energy is conserved [55].

4.3.1 Iteration loop structure

In a time dependent solve of the equation, a simple, explicit, finite difference in timestep loop is the outermost loop. There are also loops over the other dimensions in the problem: a loop over energy groups, a loop over octants and a loop over angles within the octant, and a loop over space which respects the data dependency of the sweep. Additionally there are the iteration loops: the inner and outer iterations. The inner iterations are the iterations on the

```
for Time-steps do
    for Outer iterations do
        Update source: fission term
        for Inner iterations do
            Update source: external and scattering terms
            Perform a sweep
                                        ▷ i.e. invert collision-streaming operator
            Check inner convergence of $\psi$
        end for
        Check outer convergence $\psi$
    end for
end for
```

Figure 4.2: Transport equation iteration overview (from [24])

source. The outer iterations form either another part of the iterations on the group-to-group scattering source or some form of power iterations to work out eigenvalues associated with a partitioning of the operators in the equation. The iteration structure is summarised in Figure 4.2 with a flow diagram presented in Figure 4.3.

In the case of SNAP, the proxy application used in this thesis which will be introduced in Section 4.5, the inner iterations update the fixed source and the within-group scattering — the part of the group-to-group cross section integral where the groups are the same; i.e. $\sigma_s(\vec{r}, E' \to E', \hat{\Omega}' \cdot \hat{\Omega})$. The outer iterations perform the group-to-group scattering between different groups; it is in this step that the Jacobi or Gauss-Seidel iterations occur.

4.3.2 The sweep

For a given angle in the domain, neighbouring cells in the mesh can be categorised into *upwind* and *downwind* depending on whether the solution on the cell boundaries is already known. In a non-boundary cell and for a chosen angle in the S_n discretisation, the cells downwind of a particular cell are those cells which require the solution of this cell in order to determine their solution. The upwind cells are those cells which must be solved previously in order for the cell in question to be solved itself. This is shown pictorially in Figure 4.4.

The angular direction through the mesh determines the neighbouring upwind cells, and hence defines the total data dependency across the mesh of cells. The data dependency may be described for a structured Cartesian mesh as a DAG. In an unstructured mesh the upwinding dependency may produce a cycle in the dependency graph, which poses extra challenges to the solution which are beyond the scope of this thesis.

In a structured mesh, the data dependency is the same for all angles in the same quadrant/octant. The sign of the angular cosine is also equal to the sign of the step in the spatial domain iteration. The steps are shown in Table 4.2 which describes the loop index update in the serial loop over the spatial domain in each dimension. For the first octant therefore, the first cell solved is (nx, ny, nz), and hence the boundary solution must be known along these external edges of the cell. The sweep then proceeds through the cells until

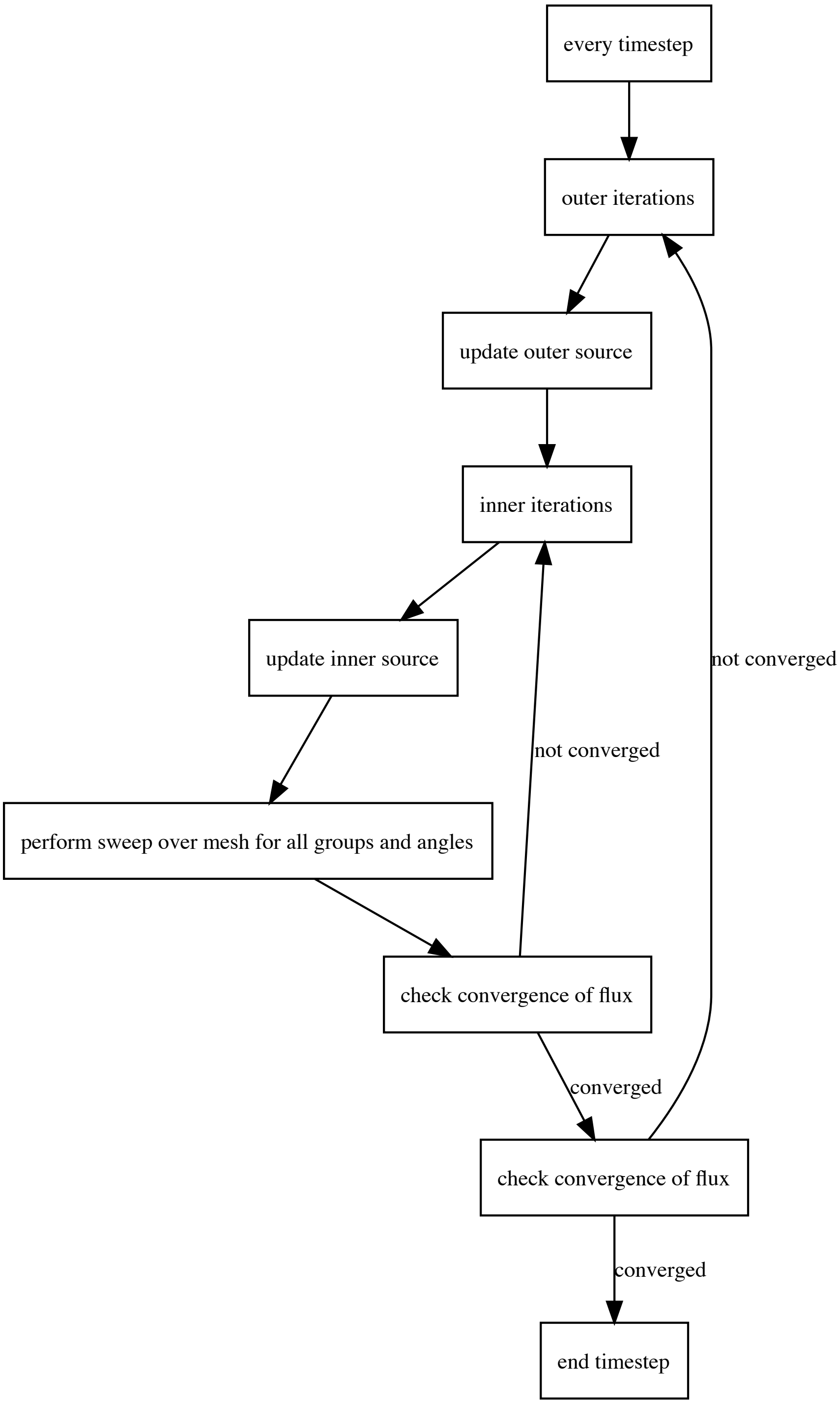


Figure 4.3: Flow diagram of iteration structure of the solution of the transport equation

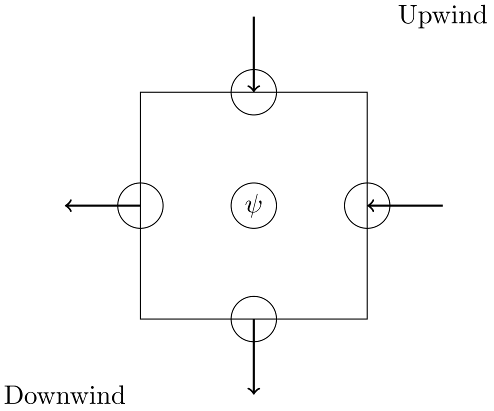


Figure 4.4: Upwind and downwind cell boundaries (from [24])

Octant	x step	y step	z step
1	-	-	-
2	+	-	-
3	-	+	-
4	+	+	-
5	-	-	+
6	+	-	+
7	-	+	+
8	+	+	+

Table 4.2: Cell ordering for octant sweeps

the first $(1, 1, 1)$ cell is reached centred at the origin. For this octant, a general cell (i, j, k) (with $i < nx, j < ny, k < nz$) has a dependency on the three cells $(i + 1, j, k)$, $(i, j + 1, k)$ and $(i, j, k + 1)$.

4.3.3 Negative flux fix-up

In the solution of the transport equation, it may possible for the angular fluxes to become negative and affect the stability and validity of the solution. Whilst this is a valid solution in terms of the numerical scheme used to solve the transport equation, it is nonphysical and therefore is usually addressed in the application. Fluxes may go negative as a result of the mesh resolution being too coarse or large in magnitude cross sections [55]. With a coarse spatial mesh the average distance travelled by the neutral particles (the mean free path) may exceed the size of the cells, and as such the movement of particles is not captured through all cells resulting in the nonphysical behaviour in the angular flux.

In order to prevent this, simple fix-up routines are used; such a routine is used in the SNAP proxy application. The routine is non-linear and may not be run in all cells, therefore potentially creating some load imbalance. Also because of the non-linear nature with the potential for early loop exit ensuring the vectorisation of this routine is also noteworthy, although modern compilers will do it.

The fix-up routine is designed so that if the central or outgoing angular flux values become negative, they instead are set to zero. The transport equation is then re-solved in the form without the substitution of the FD upwinding dependency. That is the equation is solved for the centre with all the edge fluxes known, from either the previous solution or by setting to zero as previously described. By simply setting values to zero, information in the system is lost, and the true movement is not being modelled.

In this thesis we assume that fix-up is always turned on to ensure it remains as representative as possible to the performance of a production application. However for a sufficiently refined problem fix-up should occur only rarely, if at all.

Ray effects may also be seen with the S_n method, which although they do not produce negative solutions themselves, may highlight negativities associated with each beam of particles which would otherwise not be apparent in the scalar flux.

4.3.4 Boundary conditions

The upwinding during spatial discretisation requires that cells which are positioned on the edge of the mesh are supplied with appropriate boundary conditions for their incoming edge angular flux. Setting this incoming data always to be zero is known as a *vacuum boundary.* It is also possible to specify a *reflective boundary*, where outgoing particles are reflected back into the spatial domain, as if there were a mirror at the boundary. Reflective boundaries are often used as an optimisation for symmetrical domains to reduce the size of the mesh required to simulate.

In order to begin the sweep across the mesh, boundary conditions for at least two adjacent faces of the domain must be known. Reflective boundary conditions impose an ordering on the octant sweeps, whereas with all vacuum boundaries the octants may be swept in any order. For simplicity of implementation, vacuum boundary conditions are always assumed throughout this thesis, although no optimisations are made using this knowledge. This thesis focuses primarily on techniques for sweeping a general octant in a pipelined manner, and as such does not impose any ordering nor exploits this fact. Therefore the work in this thesis can be applied independently of the boundary conditions.

4.4 Other solution approaches

Solution of the transport equation (4.1) by the deterministic method so far discussed in this chapter is only one such approach. In the Astrophysics and reactor communities it is common to use a Monte Carlo or ray tracing method (Method of Characteristics). The review paper of Bisbas et al. details 12 astrophysics codes [18], of which the majority use a form of ray tracing, with the remainder using a Monte Carlo approach. An example of using ray tracing in astrophysics is the `SMMOL` code [84]. These alternative methods will be briefly discussed to provide a wider context to the work on deterministic S_n transport presented in this thesis.

4.4.1 Monte Carlo transport

Rather than formally discretising (and therefore approximating) each of the problem domains and employing an iterative scheme to approximate the solution to the transport equation, a Monte Carlo approach can be used. A number of particles are tracked as they move through the domain, with the cross sectional data forming a probability distribution which can be randomly sampled in order to model the interactions of the particle. The interested reader is referred to Lewis and Miller for more theoretical details [55].

A Monte Carlo approach may allow for more accurate physical solutions than the deterministic approach. The spatial domain is typically represented using constructive solid geometry although it is also possible to discretise to form a mesh. The energy domain often uses continuous data rather than discretised into energy groups and so the discretisation approximation is not introduced. Additionally, floating point inaccuracies can be attributed to statistical noise in the sampling distribution, rather the directly causing round-off issues in a deterministic method. However, sufficient numbers of particles need to be present across the domain in order to evaluate statistically robust results. Where materials designed to stop particles such as the shielding around fission or fusion reactors are present in the problem domain it may be challenging to generate sufficient particle histories outside the shielded region. Also, some form of variance reduction is required, which adjusts the probabilities to propagate particles longer distances and compensating for this by varying the particle weights. Using a deterministic solution method does not carry such issues, as a solution will be found in all regions of the domain; however other issues such as ray effects may occur (see Section 4.3.3). The Monte Carlo method may be combined with the deterministic approach in order to accelerate the convergence of the Monte Carlo solution [17, 45].

The on-node parallelisation of the Monte Carlo method has been investigated by Martineau and McIntosh-Smith [59], who found that following the full history of each particle (before following the next particle) gave the best performance in comparison to evaluating each possible interaction in turn. The authors also found that it was the memory latency hiding properties of GPU architectures which enabled the platforms to give the best performance out of those tested.

4.4.2 Method of Characteristics

The Method of Characteristics is a general method to solve partial differential equations by transforming them into an ordinary differential equation along a single spatial coordinate: the characteristic curve. In general, a point in the spatial dimension may be found by traversing along a direction by some distance from a defined point (such as the origin). A number of such points are chosen and these *characteristic lines* (or tracks) are projected through the domain from them. By differentiating the transport equation by offsets along a characteristic line and using an integration factor, the angular flux at any point in the spatial domain can be found based on the flux at the origin and a source term.

The discretisation of the angular domain using discrete ordinates and the energy domains using the multi-group approximation for the Method of Characteristics is similar to that employed with the deterministic approach explored in this thesis. The interested reader is recommended the derivation in the Open-

MoC documentation [61, 62] and the treatment of integral transport methods in Lewis and Miller [55].

The main kernel of the Method of Characteristics applied to the transport equations is called a sweep, which evaluates the scalar flux along each of the characteristic lines in a manner akin to ray tracing. It is different to the sweeps employed in the deterministic S_n approach which involved sweeping through the spatial domain along a single angle (recall Section 4.3.2). A sweep in the Method of Characteristics follows each of the characteristic lines through the domain [63], and as such each line only touches a subset of the spatial domain (those which lie under the track); this is in contrast to the S_n sweep which involves the entire spatial domain.

This method is commonly applied only for a 2D domain for stationary problems [38]. In this thesis, the focus will be on 3D spatial domains, under a time-dependent regime. Additionally, codes such as OpenMoC use constructive solid geometry to represent the domain rather than a mesh based approach [61], as in the majority of Monte Carlo methods. The mini-app exploring this method in three spatial dimensions, SimpleMoC, appears to be somewhat floating point operation (FLOP) bound as the developers found that it achieved 60% of LINPACK performance but only 5% of STREAM memory bandwidth [38]. Kochunas and Downar have developed a performance model for the Method of Characteristics, which shows that the kernels have low computational intensity, and as such should be memory bandwidth bound under the Roofline model [51]. This is in contrast to the findings of Gunow et al. , which found that SimpleMoC achieves a very low fraction of memory bandwidth [38] and therefore implies that further optimisation and characterisation of the method is required. Kochunas and Downar develop the model to consider distributed memories and conclude that network latency is an important constraint on the performance at scale [51]. In Chapter 7 it will be shown that network latency is also an important architectural factor in the performance of deterministic S_n transport.

4.5 The SNAP mini-app

The SNAP mini-app from Los Alamos National Laboratory (LANL) was released as a performance proxy for transport applications [97]. The code is a simplification of a 'production code', and as such SNAP deliberately solves no real physical problems; the data is auto-generated from simple rules and has no physical meaning. There is none of the complexity of a production code. As such it is of no use for answering physical questions. However it contains a similar computational load and communication pattern to the production code, and therefore the performance at the node level and at scale can be usefully investigated within this simplified code base.

Within SNAP there are inner and outer source updates and a sweep across the mesh solving a simplified version of the transport equation.

SNAP has time dependent and stationary (steady state) solution modes, however we consider mainly the time dependent solution as this provides additional challenges that this thesis seeks to address.

SNAP is written in around 4000 lines of Fortran and uses Message Passing Interface (MPI) and OpenMP for parallelism. The MPI is used for spatial decomposition, and OpenMP threads are used for parallelism across groups.

Further details about the parallel scheme will be shown in Section 5.1.1.

The use of mini-apps for research into the performance of production codes is an established practice. Many of the US national laboratories produce suites of applications for this very purpose, and they are often used as benchmarks for the procurement of their large systems, including the upcoming Alliance for Application Performance at Extreme Scale (APEX) project which includes the procurement of two large machines, Crossroads and NERSC-9, in 2020.

The United Kingdom Mini-App Consortium (UK-MAC) have released a series of codes which have been used to help develop and understand the performance of new and existing algorithms on multi- and many-core devices. These algorithms include Hydrodynamics and Linear Solvers, and are released as CloverLeaf, TeaLeaf and BookLeaf.

The Mantevo Benchmark Suite is an effort by Sandia National Laboratory (SNL) to provide a suite of mini-apps for a similar purpose [42]. These applications have been used extensively as part of the United States 'co-design' effort with a variety of hardware vendors.

4.6 Other transport and sweep based mini-apps, benchmarks and applications

The sweep data dependency or wavefront pattern is not unique to the solution of deterministic transport and this pattern appears in other guises.

4.6.1 Dynamic programming

Dynamic programming is one such example of an approach which often involves a wavefront pattern. Dynamic programming relies on a recursive solution of partitions of the original problem which can then be combined. This generates a DAG which can be solved satisfying the dependencies in parallel with a wavefront.

A solution for accelerating dynamic programming algorithms using GPUs has previously been proposed [58]. This method breaks the dependency between wavefronts with a post-processing step to correct the solution; however, it relies on a linear relationship between the wavefronts and as such this approach is not applicable to transport sweeps.

The Needleman-Wunsch algorithm from the field of genetics is used for sequence alignment and it is typically solved using dynamic programming approaches. An accelerated solution, in particular using GPUs, has been tried by Krommydos et al. although that study noted that the GPU provided no speedup over the original CPU implementation [53]. The authors of the study suggest no reasons as to why this is the case.

Edit-distance is another such algorithm which when solved with dynamic programming utilises a wavefront. Parallel implementations of the Wagner-Fischer algorithm for edit-distance use a blocking technique whereby diagonal blocks can be solved concurrently, thus increasing the work to do between synchronisations between levels in the wavefront [15]. This blocking approach could be applied to a transport sweep however the solution of the transport equations offers extra dimensionality beyond space (over the edit-distance data) which is available for parallelism; notably angles and energy groups which may be blocked in a similar manner to increase the computational load between synchronisations of a wavefront.

Although dynamic programming on the surface exhibits a wavefront style sweep, the complex nature of the nested iterative solution required to solve the integral-differential transport equation makes the fields very different. As such any proposed solutions to dynamic programming at large scale or on GPUs are not applicable to the topic of this thesis.

4.6.2 Lower-upper matrix factorisation

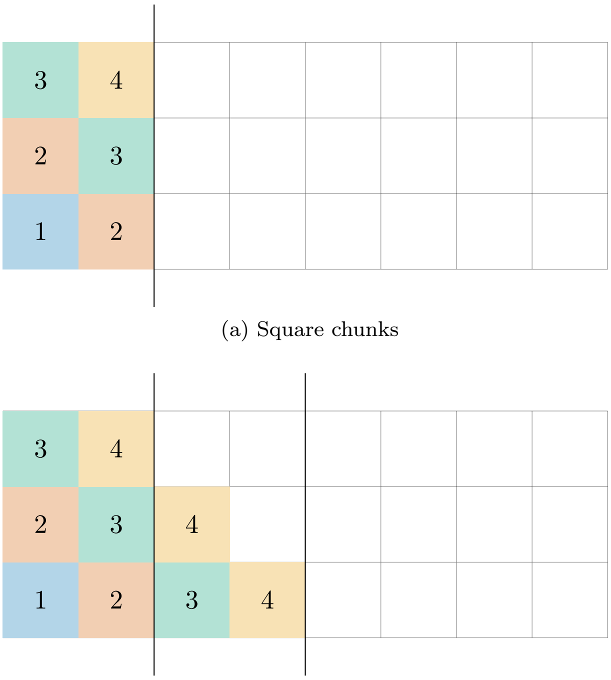
A lower upper (LU) Gauss-Seidel solver is included in the NAS Benchmark Suite [8]. Somewhat similar to true LU factorisation, the matrix is split into the sum (rather than the product) of an upper and lower triangular matrix. The Gauss-Seidel approach allows the linear system to be solved iteratively, rather than directly, using forward substitution. The forward substitution results in a wavefront style data dependency.

A GPU implementation of the NAS LU benchmark has been explored in CUDA by Pennycook et al. [80]. CUDA threads are mapped to cells in each hyperplane in the wavefront, with a new kernel launched between each hyperplane to satisfy the data dependency. This showed good performance improvements over the CPU benchmark from the use of the GPU. Pennycook et al. also highlighted issues with scaling this wavefront algorithm beyond a small number of GPU accelerated nodes.

The LU solver is very simple compared to a transport sweep. Firstly, there are only two sweeps, whereas a 3D transport code requires eight per iteration. Secondly, it lacks the additional problem dimensions of transport (angle and energy group) and as such only has the three spatial dimensions available for parallelism. The hyperplane parallelism approach does however leave the GPU underutilised for a proportion of the sweep; initially only one cell can be computed. The GPU remains underutilised until the size of the wavefront is large enough to saturate the many-core device with useful work from spatial parallelism alone.

Pennycook et al. do however give a modification to the standard orthogonal chunking of the work in a 2D domain decomposition (as will be described in Chapter 7) when employing the local spatial wavefront parallelism. Rather than divide each pencil (spatial subdomain, see Glossary and Chapter 7) into square chunks for communication resulting in a start up and tear down cost for each chunk, their suggestion is allow the full hyperplane width for as much of the pencil as possible. This is shown in Figure 4.5 where communication occurs after two planes, where the bold vertical lines indicate communication. The colours (and numbers) show which cells are operated on for each stage of the sweep. In Figure 4.5a the maximum spatial parallelism (3 cells) within the pencil is never reached. Whereas in Figure 4.5b once the width of the pencil reaches 3 it remains at 3 until the end of the pencil resulting in only a single start up and tear down to this maximum width. Additionally, about half of the next chunk is already computed.

Unfortunately this cannot be applied as a simple modification to the transport algorithms used throughout this thesis. The issue surrounds the unavailability of boundary data for subsequent chunks after the first chunk for subdomains with internal boundaries. Using the illustration in Figure 4.6, this first



(a) Square chunks

(b) Diagonal chunks

Figure 4.5: LU chunking options adapted from Pennycook et al. [80]

rank may communicate the two cells at the top edge of the first chunk to the neighbouring rank, after completing four wavefronts. As this rank is on an external boundary, there is known boundary data for wavefronts 3 and 4 which enter the second chunk and the wavefront may extend beyond the orthogonal chunk size. The neighbouring rank however has not yet received the boundary conditions for the second chunk and so is unable to compute all cells on the diagonal for wavefronts which go beyond the chunk boundary without stalling the compute to check for this receipt of these values from a second communication; such cells are notated '?' in wavefront 7.

One solution would be to send data each time a boundary cell was computed, however the number of messages would increase and the messages themselves would be very small consisting of data for only a single cell. This is contrary to original purpose of chunking as an optimisation strategy, which sought to better utilise the network by sending fewer, larger messages.

This issue may become mitigated by specialising the compute of the first and last chunks of each sub-domain on the rank where the sweep begins for wavefronts which do not include an internal boundary cell. Therefore the strategy of using diagonal chunks has not been investigated further due to its complexity of implementation within the SNAP proxy application. Additionally, small chunk sizes were shown to provide the best performance [25], and so the diagonal chunking approach provides little additional benefit in this case.

An optimisation study of the HPCG benchmark, which contains a Gauss-Seidel sweep, suggests using a task dependency graph for the sparse matrix-vector multiplication and synchronising between parent and child tasks rather

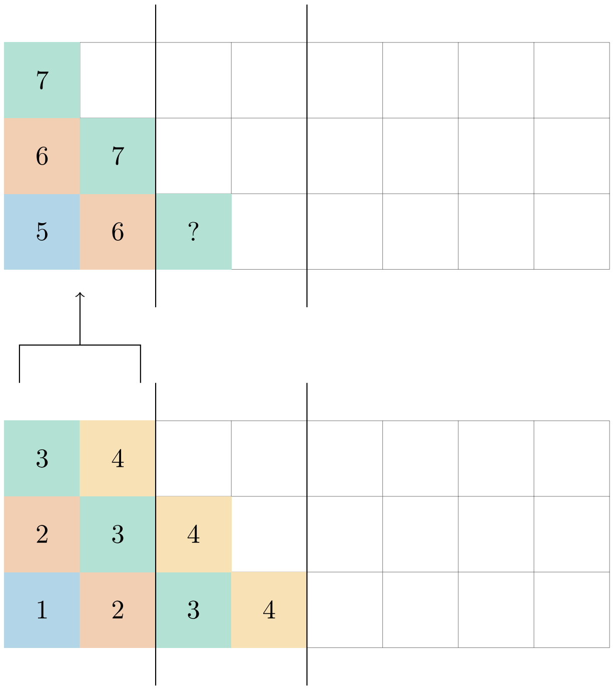


Figure 4.6: Communication issues in the LU chunking option of Pennycook et al. [80]

than global barriers for multi-core execution, and multi-colouring for many-core execution [77]. The construction of the DAG for transport and the local synchronisation scheme may be attractive for unstructured transport but would provide little benefit in the structured mesh case. However the colouring scheme is not applicable to transport as there is no re-ordering to reduce the data dependency in contrast to sparse matrix operations where this is a valid approach.

Another approach used blocking schemes for GPU acceleration of a dense Gauss-Seidel solve to reduce the number of device global synchronisations [20]. This approach does not transfer to a transport sweep as the technique is designed to work in a constraint solving environment on a dense matrix; the matrix for transport, although never constructed due to its sheer size, is a band matrix and so would be considered sparse with only the diagonal along with 2–3 off-diagonals populated.

4.6.3 Sweep3D

The Sweep3D proxy application was developed at LANL. It solves a single energy group of the multi-group equation, with no support for computing multiple groups. It utilises the S_n algorithm but does solve all the angles in the quadrate set. Sweep3D only uses MPI for parallelism in its implementation. This benchmark is no longer available and is all but superseded by SNAP.

4.6.4 KRIPKE

The KRIPKE mini-app from Lawrence Livermore National Laboratory (LLNL) is designed to be a proxy for ARDRA. It is written in C++ and simulates the S_n transport algorithm on a structured grid, with multiple energy groups. Its original focus was to investigate the data layout for the angular flux along with the appropriate loop ordering which produced the most performant code. This was done manually by implementing all the possible orderings as separate kernels. Automatic compiler vectorisation was performed on the inner loop, and OpenMP parallel threads were applied to the energy group loop. This application of threads was always at this domain level and so may not have always been appropriate. The MPI implementation is based on the Koch, Baker and Alcouffe (KBA) algorithm [50], as is SNAP (more details on this will be found in Section 7.1).

The original paper demonstrated the applicability of the benchmark in terms of a performance proxy for ARDRA [54]. There has been later work on porting and benchmarking the application to a variety of architectures including GPUs.

A port of KRIPKE to GPUs by Appelhans suggested that some spatial tiling might be possible within the KBA sweep [7]. The sub-domain would be further divided into pencil-shaped sections, computing each section in turn. As such the synchronisation between wavefronts of cells need only be local (not device wide or with the host) and some outgoing angular flux data need not be written to device memory thus reducing the number of memory operations. The investigation was limited to local cubic domains, which would correspond to a single chunk of the sub-domain. However this approach would require a large spatial sub-domain; something which is most unlikely given the standard decomposition and memory capacities of GPUs. The investigation used a cube with sides of 32 cells which is many more cells than a spatial sub-domain would normally consist of under the KBA algorithm. Also the spatial decomposition does not result in local cubic shaped sub-domains, and as such the spatial size is much more limited, meaning that even using the smallest aggregation of cells would result in very little parallel work. Whilst the over-decomposition scheme of Adams et al. may result in multiple cubic domains on a node [2], the footprint for multiple 32-cube sub-domains would be much too large. KRIPKE lacks realism as a mini-app for transport applications by only considering a cubic spatial domain run on a single node. For example it is more realistic to consider a chunk of the sub-domain being a cube with sides consisting of 4 cells. Therefore the scope for tiling in the manner proposed by Appelhans is severely restricted, and results in the maximum number of independent units of work expressed in the kernel being reduced in comparison to the scheme proposed as part of this thesis in Chapter 5.

In practice KRIPKE could be considered an equivalent to the SNAP benchmark as they solve similar equations in a similar manner. KRIPKE is time independent (stationary), yet does store both the angular and scalar flux; storage of the angular flux is not necessary for a stationary calculation.

All of the conclusions in this thesis can be directly applied to KRIPKE.

4.6.5 Denovo

Denovo is a full application for reactor assembly calculations from Oak Ridge National Laboratory (ORNL) [32]. The application is not publicly available. As with SNAP and KRIPKE, it uses the KBA algorithm on a 3D Cartesian structured mesh. Rather than iteration of the scattering source, a Krylov (GMRES) method is used instead. Denovo is not time-dependent and therefore does not store the angular flux, only requiring storage for the scalar flux (and angular moments for anisotropic scattering). The sweep routine consumes the majority of the runtime, on the order of 80%. Denovo is a hybrid MPI and OpenMP application, with OpenMP threads used for angles and energy groups. Octants are fully decoupled based on assuming vacuum boundary conditions on the problem exterior and so all angles and octants are treated independently. As previously stated, this thesis does not assume vacuum boundary conditions so that the conclusions will be applicable to all settings.

The GPU port of Denovo uses CUDA, however the performance comparisons by Baker et al. are made with respect to only utilising half of each Titan node [10]. For the sweep routine they demonstrated a 4–6X speedup on Titan over its CPUs, having previously shown a 3.5X speedup on Jaguar. The application as a whole showed a 2X speedup on Titan utilising the GPUs. The energy groups and octants are decomposed across MPI ranks artificially inflating the node count, and thus involving additional all-to-all communications within processor subsets. As such this CUDA port of Denovo shifts the balance of the application away from the sweep to other parts of the application as a result of the increased communication costs due to the choice of decoupling and decomposing the octants and energy groups in the manner described. A nested KBA sweep is implemented at the thread block level on the GPU. The work is also partitioned at the warp level where scalar flux moments were computed by each warp. As Denovo is time independent, and does not store the angular flux, it does not become memory capacity constrained on the GPUs, which have limited memory capacity compared to the host CPUs.

As Denovo is focused on a stationary solution it remains of limited interest and applicability for this thesis. Additionally, as it is closed source it is not available to work on directly.

4.6.6 UMT2013

The UMT2013 proxy application from LLNL is designed to be a proxy for a 3D unstructured S_n radiation transport calculation. The code is written in a mixture of Fortran, C and C++, and uses a hybrid of MPI for spatial parallelism and OpenMP for parallelism of angles. This is a rather large mini-app at around 50,000 lines of code and the documentation is also rather lacking. As such it is an impractical research vehicle.

The benchmark is an update of the UMT and UMT2k benchmarks. It uses an upstream corner-balance spatial discretisation in 3D and as such the solution method itself is different to the FD (and later finite element method (FEM)) approaches and so it is not appropriate to the focus of this thesis.

4.6.7 Tycho 2

The Tycho 2 mini-app is a S_n sweep on tetrahedral unstructured meshes. It uses linear discontinuous Galerkin (DG) finite elements. The sweep dependency of each angle generates a DAG, and it is the traversal of this graph that the application focuses on. This work continues that of Pautz [78]. It investigates different ways to perform the sweep in parallel on a distributed machine.

There are three such methods currently implemented. Firstly, a standard sweep routine somewhat similar to KBA but modified to work on unstructured grids.

A parallel block Jacobi routine is also implemented, where each sub-domain is iterated on in turn with out-of-date boundary information, with the sub-domain still operating a local sweep routine. A standard halo exchange is performed to update the boundary data. The application developers admit that this takes longer to run than the standard sweep, presumably because of the greatly increased iteration count. However they suggest that the halo exchange routine scales better than the standard sweep scaling, which is clearly true given the findings of Chapter 7, however they provide no results of the scaling of this method. They also provide no information on time to solution compared to the standard sweep, however this approach is pragmatic to the scheduling of an unstructured mesh sweep. This approach should also remove the start up and tear down costs of the sweep.

The final routine is based upon the Jacobi solve, however it uses a Krylov solver to accelerate the lagging of the boundary data.

The application focuses on the parallel regime to solve the equation, rather than the overall efficiency itself. Particularly, the construction and solve of the small linear systems at the centre of the FEM are not the focus of this application. This is also a topic of this thesis as explored in Chapter 8.

The application is written in C++, and comes with a variety of auxiliary programs to generate and partition the mesh. These auxiliary programs either implement a naive and simple partitioning, or else impose a dependency on an external library. As such in their simplest, self contained form, it cannot be a true proxy of a production app, which probably does not do a naive mesh decomposition.

Initial results from Garrett suggested that Tycho 2 running on Intel Xeon Phi (Knights Landing) (KNL) achieved around parity performance in terms of runtime with a dual-socket Xeon processor [34]. The arguments presented to justify that this is the expected performance are flawed; they suggest that the KNL has twice as many cores at half the speed compared to Xeon, and so should achieve around parity performance. Both total floating point operations per second (FLOPS/s) and total memory bandwidth from Multi-Channel DRAM (MCDRAM) on the KNL are much larger than the theoretical peak performance of a dual-socket Xeon — whatever headline metric is used the KNL should in theory outperform Xeon. Therefore, if this benchmark achieves parity performance there is a performance issue to be investigated by the authors.

Tycho 2 solves multiple energy groups in parallel, using OpenMP, however there is no group to group coupling. The authors of the code also claim that the code requires optimisation, in particular for memory accesses [34].

In summary, this benchmark is too immature to be used in this thesis. Furthermore, the primary focus of work contained within this thesis is also on structured grids, and so as this benchmark operates on an unstructured grid the assumptions inherent in the structured case are not exploited sufficiently to obtain equivalent levels of performance.

4.7 Summary

The solution of the transport equation as detailed in this chapter presents a number of unique challenges compared to many other Boltzmann equation solvers or codes which exhibit a wavefront pattern. It is a highly dimensional problem, with different levels of concurrency available in each of the dimensions. Most notably a sweep across the spatial domain is required, and it is not possible to compute the solution for each spatial cell in parallel as in a typical concurrent scheme.

The solution scheme is implicit, requiring that a number of iterations are performed for each timestep; this is in contrast to an explicit time dependent solver of fluid dynamics codes where the solution at the next timestep can be calculated with a single update. Also there is a large memory footprint associated with the solution itself as a result of the high number of dimensions. Finally, the iteration structure of the solver requires deeply nested loops formed from both convergence iterations as well as loops over the domain space. Although not the only algorithm to exhibit a sweep dependency, the extra complexity of the additional problem dimensions means that much of the previous work on sweeps is not applicable to transport.

Mini-apps will be used as a research vehicle in order to practically investigate the solution of the transport equation on different computational devices. These applications remove much of the physical meaning of the solution by using auto-generated and simplistic fictional data. As such they focus on the algorithm itself rather than the complexities of production applications where a large number of options and complex data initialisation routines occur. The mini-app therefore focuses on the loop structure, communication and data access patterns of the algorithm. The resulting code base is much smaller than a production application and is therefore much more agile for research purposes. The SNAP mini-app from LANL will be used extensively throughout this thesis. The development of new mini-apps as part of this thesis will allow for more focused investigation into specific aspects of the algorithm, and will complement SNAP. Chapter 6 describes the author's work on a mini-app designed specifically to understand performance issues observed on cache based architectures.

CHAPTER 5

Accelerating transport on GPU architectures

The work in this chapter also appears in the following publications:

- Tom Deakin, Simon McIntosh-Smith and Wayne Gaudin. *Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport.* Workshop on Representative Applications at IEEE CLUSTER, 2016.
- Tom Deakin, Simon McIntosh-Smith, Matt Martineau and Wayne Gaudin. *An Improved Parallelism Scheme for Deterministic Discrete Ordinates Transport.* International Journal of High Performance Computing Applications (special issue), 2016.

The SNAP proxy application from Los Alamos National Laboratory (LANL) is a parallel code and as such has a specific mapping of the algorithm to the various parts of hardware such as vectors and programming model abstractions such as threads. The mapping of the algorithm to hardware is important to ensure good performance is achieved. The parallel scheme of SNAP will be introduced in this chapter. A concurrent scheme will then be presented to enable good performance of the proxy application on GPUs by leveraging as much of the available concurrency in the algorithm as possible and mapping appropriately to the programming model abstractions used for programming such devices. This scheme will then be tested and shown to provide significant speedups on GPUs for the first time for the SNAP application, despite previous efforts to port SNAP to GPUs. The memory bandwidth of the kernel will also be modelled to show that these speedups are in line with the expected performance gains for a memory bandwidth bound kernel.

The domain decomposition of the SNAP mini-app will be discussed in detail in Chapter 7, however it is sufficient to consider for now that each process will receive a spatial sub-domain of the full problem. The performance of solving the local sub-domain is investigated in this chapter.

5.1 Parallelism in the SNAP mini-app

5.1.1 Original scheme

The SNAP mini-app uses a hybrid programming model of Message Passing Interface (MPI) and OpenMP in which to express the parallel work. The original SNAP code is CPU only and does not support GPUs. MPI is used for spatial decomposition according to the Koch, Baker and Alcouffe (KBA) algorithm [50], although the details of this are deferred until Section 7.1.

The full angular domain is grouped into octants, and due to the regular, structured Cartesian grid all angles within one octant are independent of each other. A sweep over the spatial mesh in SNAP is for a batch of *all* angles within the octant, rather than a sweep for a single angle at a time (as originally conceived with the KBA schedule). SNAP uses compiler auto-vectorisation to compute the solution for angles in one octant in the cell in parallel. The compiler generates Single Instruction Multiple Data (SIMD) vector instructions based on the underlying instruction set of the CPU architecture.

The boundary conditions in SNAP are always vacuum; that is the incoming fluxes across the boundary to the mesh are always zero. This therefore means that there is no dependence between angles in different octants. However, SNAP does not utilise this parallelism to compute all angles concurrently; and therefore this is not considered further as a potential source of concurrency. In part, this is a design decision made to ensure that SNAP remains representative of PARTISN for which it is a performance proxy. Additionally, not exploiting this parallelism ensures that results will be applicable to a wider array of transport codes and problems which may have more complex boundary conditions such as reflective boundaries. The octants are therefore treated as a serial iteration, performing a sweep for each one in turn. Note that reflective boundaries do not entirely preclude this as a source of parallelism in general, although it would become a problem dependent source of parallelism depending on the specific choices of boundary properties.

SNAP considers energy groups in the energy domain to be independent as a result of using a Jacobi iteration scheme in this domain, with the group-to-group coupling taken into account in the definition of the source terms [13]. Recall from Section 4.2.3 that such a scheme uses energy group data from a previous iteration in order to expose concurrency, but as the method converges to a tolerance the method is of equivalent accuracy to serial schemes. Therefore during each sweep the angular flux for each energy group can be computed in parallel. To express this SNAP uses OpenMP threads; however the worksharing of energy groups to threads is expressed explicitly in a Single Program Multiple Data (SPMD) style rather than utilising the worksharing constructions in the OpenMP programming model to ensure correct MPI communication avoiding deadlock. The energy groups are divided into a number of bins, with each thread assigned one bin containing one or more energy groups. In this way the sweeps for each energy group are performed independently by each thread within each MPI process. This results in MPI communications being called from within the OpenMP parallel region thus requiring high levels of thread support from the MPI library (at least `MPI_THREAD_SERIALIZED`). The original approach means that when considering a version of the code which supports GPUs, much of the MPI needed refactoring in order to work with the offload model used for GPU

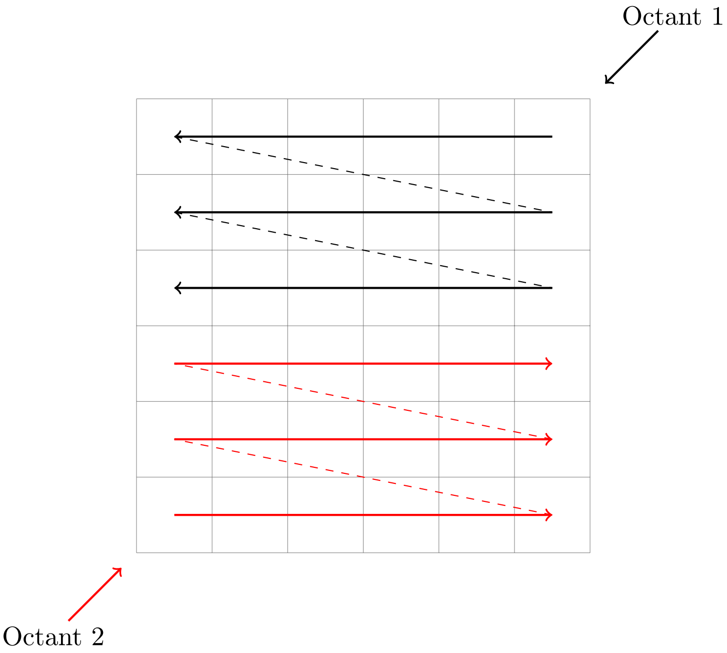


Figure 5.1: Serial sweep for two octants (from [24])

applications. This approach may perhaps be viewed as distributing the energy groups across different processors in a manner similar to the Denovo code (see Section 4.6.5) [32]; the first core assigned to each MPI process computes the first energy group. However the decomposition is via OpenMP which means group-to-group communication in the source term is local to each process rather than across the network as in Denovo.

The sweep itself is performed serially in the spatial domain, as shown in Figure 5.1 (note that only one octant is computed at any time). By iterating serially over space the data dependency of the sweep is maintained ensuring correctness.

Note that CPU hardware resources are fully allocated with this scheme despite a serial loop in space; something contrary to many other High Performance Computing (HPC) applications where the parallelism comes entirely from the spatial domain (for example Lattice Boltzmann codes [69]). A thread is launched for an energy group, with this thread then iterating over each cell in the subdomain sequentially using vector instructions to compute all the angles within each cell. Therefore threads and SIMD instructions fully utilise the available levels of parallelism on CPU architectures (instruction level parallelism is not explicitly programmed and comes from either the compiler rearranging instruction streams or an out-of-order execution pipeline in the architecture).

5.1.2 Concurrency for many-core

The original parallel scheme of Section 5.1.1, whilst providing sufficient concurrency for multi-core CPUs does not provide enough concurrent work for many-core architectures, in particular a GPU. Therefore additional parallelism

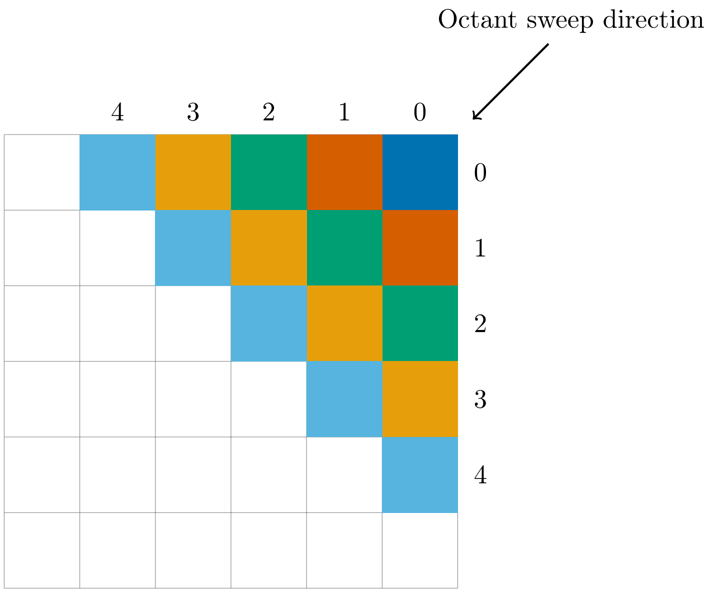


Figure 5.2: Wavefront sweep across a 2D grid (from [24])

must be found and expressed so that many-core technology may be exploited.

The concurrency in the angular and energy group domain is already fully exploited by the original scheme. Again we are restricted to running a single octant at once so that dimension is not considered as a source of concurrency. The spatial dimension however exhibits a natural parallelism between cells on each wavefront. This is shown pictorially in Figure 5.2, where cells are coloured along each wavefront.

All cells within a wavefront (those of one colour) only depend on incoming flux values from cells in the previous wavefront; there is no dependence on cells in subsequent wavefronts. Therefore cells along each wavefront can be computed in parallel, ensuring synchronisation between wavefronts so that the upwinded data dependency is respected. This synchronisation is shown as dashed lines in Figure 5.3.

The concurrency available from cells on a wavefront can be combined with the parallelism within each cell in the angular and energy domain. However note that there is no further additional parallelism available from the Discrete Ordinates (S_n) algorithm and this scheme fully uses the potential concurrency. It is important to consider this here as there may be physical problems specified by choice of mesh, quadrature and material data, which reduce the available parallelism; examining the best attainable performance on many-core for the general algorithm is important.

A *unit of work* is defined to be the calculation of the angular flux for one octant sweep o in a given cell (i, j, k) for angle a and energy group g. In this new parallel scheme each cell therefore has

$$N_{\text{angles}} \times N_{\text{groups}} \tag{5.1}$$

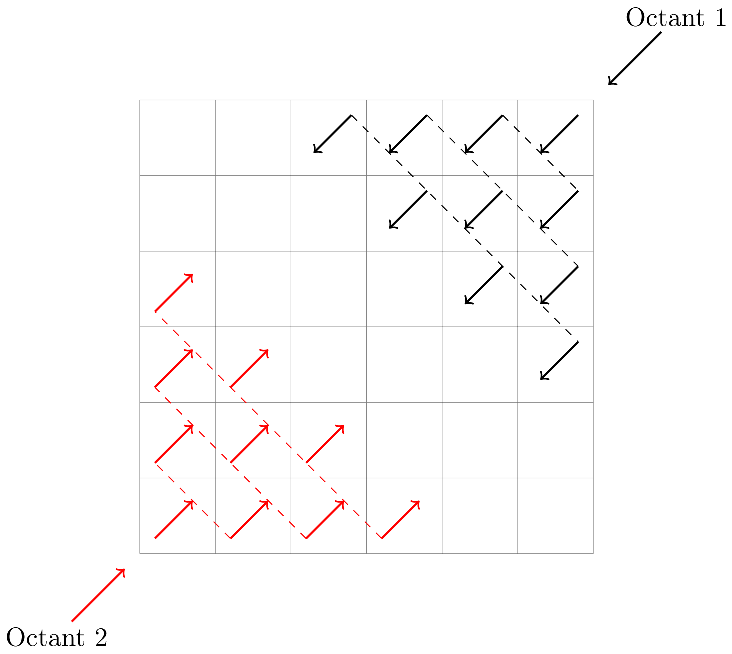


Figure 5.3: Spatial parallel sweep for two octants (from [24])

concurrent units of work. There are at most

$$O\left(\frac{N_x N_y N_z}{\max\{N_x, N_y, N_z\}}\right) \tag{5.2}$$

concurrent cells in each wavefront. This is determined by the product of the two smallest spatial dimensions in a three dimensional grid. The total available parallelism is then the product of (5.1) and (5.2).

Note that during the sweep the maximum number of cells as expressed in (5.2) will not always be available. Specifically there will only be one cell at the start of each wavefront. This is clear from Figure 5.3.

As an example we consider a sample problem with 50 energy groups and the S_{32} angular quadrature which has 136 angles per octant. Therefore there are 6,800 units of work per cell (according to (5.1)). In comparison to an NVIDIA P100 GPU with 3,584 processing elements (CUDA cores), this is almost enough work from a single cell to saturate the device, assuming the rule of thumb of requiring 4 times the number of processing elements for good occupancy. As the sweep progresses, the cells on the wavefront multiply this parallelism (according to the product of (5.1) and (5.2)) and therefore there is sufficient work to schedule on many-core devices. Note too that the spatial parallelism will become more valuable in smaller problems.

Previous NVIDIA HPC GPUs such as the K40 had 2,880 processing elements; although the number of processing elements in the P100 is more than in the K40, the increase in the number of processing elements is not currently showing an exponential growth in the available parallelism from the hardware. Indeed, the latest NVIDIA GPU, the V100, has 5,120 processing elements. Therefore this scheme should provide sufficient parallelism for future GPU devices along the current trend.

It is possible to express the spatial parallelism via dependencies in the form of a directed acyclic graph (DAG). This method of programming is supported in both OpenCL and CUDA. Each node in the graph is a cell in the grid with dependencies assigned to upwind neighbouring cells. Therefore the programmer does not have to worry about optimal scheduling of work, nor be concerned if a unit of work becomes delayed due to negative flux fix-up. In this way, a kernel could specify the work in a single cell. However, when this was implemented in OpenCL using the Events Application Programming Interface (API) and an out-of-order Command Queue, there was too much expectation on the OpenCL runtime and device driver to (close to optimally) satisfy the dependencies and schedule the small size kernels on the device concurrently. Alternative API calls are available in CUDA however they provide identical functionality to OpenCL and so were not tested explicitly; note that OpenCL and CUDA performance are generally identical (for example the results in Chapter 3). The performance available with this method was not compelling enough to consider it in greater detail, and manually scheduling entire wavefront sweeps as a single kernel (the approach discussed in Section 5.1.3) gave much better performance [26].

Once the angular flux has been calculated for all angles in a cell a reduction must occur for computation of the scalar flux. This is a reduction local to each cell for each energy group and therefore does not require grid-wide synchronisation as is typical in other Boltzmann equation solvers. This is initially performed as a separate kernel once all sweeps have completed, where the angular flux in each cell was written back to memory. The original SNAP code performs the reduction in line with the computation of the angular flux, however merging the reduction in this GPU approach would require stipulating restrictions on the work-group size as well as work-group level synchronisations. The work-group size must be a power of two in order to correctly implement the reduction. Merging the kernels actually results in a longer runtime than running the sweep kernel and then a separate reduction kernel, on the order of around 35%. The ability to perform within work-group reductions efficiently while still allowing sufficient concurrent work-groups on each compute unit will be vital for future GPUs; the number of concurrent work-groups is usually limited by the size of local memory which is used for the reduction process.

5.1.3 An OpenCL implementation

As part of this thesis, this parallel scheme was implemented using OpenCL into SNAP in order to experimentally test the scheme on many-core GPUs. OpenCL was chosen as it could express all elements of the concurrency, including testing the DAG approach, in a platform agnostic manner so that GPUs from a variety of vendors could be tested. A downside of using OpenCL is that the API is C based, with kernels written in a variant of C99 whereas SNAP is a Fortran application. Therefore introducing OpenCL into SNAP required some rewriting of the kernels and iteration loops of SNAP, with a focus on ensuring that the data was resident on the GPU to avoid the large overhead of memory transfers between the host CPU and the device memory spaces.

The data layout of the angular flux was also changed to match the concurrency in the new scheme. The original data layout in SNAP was (angle, space, octant, energy); with the leftmost dimension having unit stride and the rightmost dimension having the largest stride according to the Fortran convention. This layout would result in poor memory access patterns on the GPU due to adjacent work-items in the work-group computing all angles and groups for a single cell. Therefore a new data layout was used of (angle, energy, space, octant), resulting in unit stride access for adjacent work-items in the work group for all the work in each cell. It is important that data access patterns are correct for good performance on GPU architectures.

An OpenCL work-item is assigned to a unit of work (the calculation of one entry in the angular flux array). A 2D kernel is enqueued for each wavefront of cells, with the dimensions of the kernel specifying the total number of work-items to launch. The first dimension is set to be equal to the product of angles and energy groups as in (5.1). The second dimension is set to be equal to the number of cells on the wavefront as in (5.2). The number of work-groups is not specified as the OpenCL runtime is able to make appropriate device specific choices. By launching a separate kernel for each wavefront of cells the global synchronisation between kernel calls ensures that the upwind data dependency between cells is maintained.

The scalar flux reduction is performed as a separate kernel for simplicity once all the sweeps have occurred for each inner iteration. By calculating the scalar flux in this way there are no dependencies between any cells and they may be computed concurrently. Note that the reduction only occurs within each cell. A work-group is launched per cell and energy group. The number of work-items is chosen to be the closest power of two to the number of angles so that a commutative tree-based reduction in local memory can be employed, such as those recommended by AMD [5]. The work-items within the work-group therefore reduce the angles over all octants into the scalar flux associated with a cell and energy group.

OpenCL is a cross-platform programming model, and therefore it is possible to run the same code on the CPU as well, however the focus of this approach is on many-core GPU architectures. The reduction as described above performs poorly on the CPU due to the memory access pattern and so a simpler reduction was implemented for use on a CPU which launches a work-item per cell (instead of a work-group per cell) thus performing the reduction in each cell in serial. Work-items are typically associated with threads on CPU architectures and therefore this is a more appropriate parallel strategy for this architecture than the GPU style tree-based reduction.

5.2 Performance results

In order to test the effectiveness of the improved parallel scheme on GPU architectures, the runtime of the sweeps is tested on a GPU and compared to the performance of the original SNAP application running on a dual-socket Intel Xeon (Haswell) node. The devices tested are listed in Table 5.1.

Details of the problem size are specified as:

- Grid side: $16 \times 16 \times 16$
- 50 energy groups

Platform	RAM (GB)	Theoretical peak memory BW (GB/s)	Measured memory BW (GB/s)	D.P. TFLOPS/s
Intel Xeon E5-2670 ×2	64	$52.2 \times 2 = 104.4$	60	0.33
Intel Xeon E5-2698 v3 ×2	128	$68 \times 2 = 136$	118	1.18
AMD Opteron Processor 6274	32	51.2	13	0.14
NVIDIA Tesla K40	12.288	288	194	1.43
NVIDIA Tesla K20c	5.12	208	152	1.17
AMD FirePro S9150	16	320	273	2.53

Table 5.1: Specifications of devices used for testing single node sweep performance, with measured bandwidth recorded using BabelStream (from [24])

- 136 angles per octant (S_{32})
- 2 orders of anisotropic moment expansion

The memory requirement of this problem totals 3.6 GB which is within the capacity of all the devices tested. The runtime is measured as the time taken to complete the eight octant sweeps of the mesh including the scalar flux reduction, which comprise one inner iteration.

Although this is a cubic problem and hence has an abundance of spatial parallelism with a maximum of 256 cells on the largest wavefront, the number of cells need only be a small constant factor to saturate a GPU as shown in Section 5.1.2. With other shaped domains, such as the pencils which occur with a KBA decomposition, this small constant factor should also be adequate, and the effectiveness of the concurrent scheme discussed in the present chapter will be explored in Chapter 7 in this distributed regime.

Figure 5.4 shows the performance on GPUs stated in terms of speedup compared to the Haswell baseline. The relative memory bandwidths of the GPUs compared to the Haswell baseline performance can be obtained from Table 5.1 or be referring back to Chapter 3. For a simple Triad kernel, the AMD S9150 should be 2.3X faster than the Haswell baseline according to the BabelStream benchmark. A similar improvement is seen on this device for SNAP using the improved parallel scheme here as shown be a 2.44X speedup in Figure 5.4. The ratios of attainable memory bandwidth on the NVIDIA GPUs again match the speedups obtained with the parallel scheme. Therefore the scheme does leverage the improvement in memory bandwidth on the GPU devices over the CPU to deliver the speedups in line with this advantage.

These results are also compared to those obtained by Wang et al. from their CUDA port of SNAP [94]. Their port is a mechanical reimplementation of SNAP using CUDA and does not expose any additional parallelism, utilising a similar concurrent scheme to original SNAP. A kernel is launched for each energy group with threads assigned to angles and grouped so that there is one thread block per cell. The number of concurrent thread blocks over cells on a wavefront is limited to one per streaming multiprocessor (SM) unit so that synchronisation between thread blocks can occur on the device according to the trick of Xiao and Feng [96]. This severely limits the potential concurrency and reduces the ability for memory access latencies to be hidden by running multiple thread blocks per SM. Note that the memory access is still coalesced for this scheme within each thread block as the original memory layout is not altered from the original and is appropriately matched to the parallel scheme in this case. Multiple kernels may be launched using multiple streams to leverage the energy domain concurrency, but this relied on the ability of the driver to schedule concurrent independent kernels on the hardware successfully; the DAG based implementation discussed in Section 5.1.2 similarly relies on the driver and was found not to provide compelling performance [26]. As such in the port by Wang et al. there are very few cells which are computed in parallel in practice. As Figure 5.4 shows, their approach does not successfully leverage the memory bandwidth advantages of GPUs.

Running the OpenCL implementation with the improved parallel scheme of Section 5.1.2 on the baseline CPU does result in a large slowdown. This is likely due to non-uniform memory access (NUMA) effects on the dual-socket node within the Intel OpenCL runtime. Indeed profiler output stated that there were

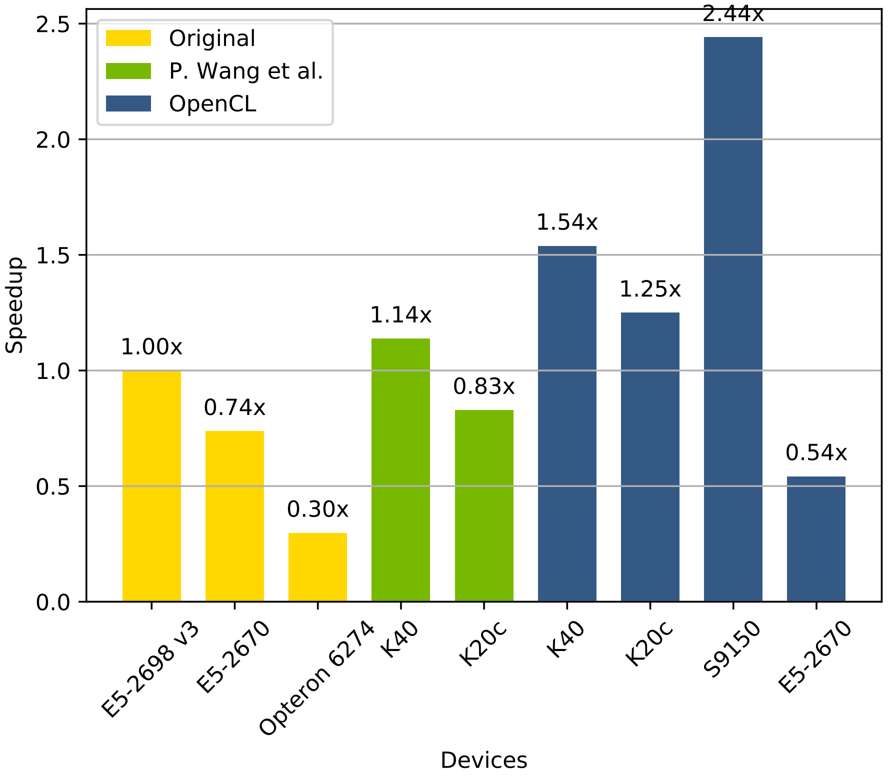


Figure 5.4: Speedup of improved concurrent sweeps on GPU (from [24, 26])

many cache misses as a result of data being in the wrong NUMA region. It may be possible to improve the NUMA behaviour by not using the OpenCL memory copy API calls and initialising the data via a kernel (as has been shown in BabelStream in Section 3.5.1), however this is not practical for the purposes of the SNAP benchmark. The Intel OpenCL runtime does not allow the user to control the number of threads launched, and so whilst running on a single-socket node would prevent NUMA issues from occurring, the common CPU-based supercomputer nodes are dual-socket and so it is the latter which provides the baseline performance here. Additionally, the memory access pattern resulting from running cells concurrently will be less optimal than the original approach implemented in standard SNAP on a cache-based architecture where the data is accessed there in a very regular fashion, with stride one access through the whole problem domain for each thread. Further study of this behaviour will be shown in Chapter 6.

5.3 Modelling the memory bandwidth

It is useful to model the memory movement of the sweep kernel so that the sustained memory bandwidth can be estimated. This then allows comparison with the gold standard Triad bandwidth so that a measure of efficiency can be derived as the ratio of these two bandwidths. The BabelStream benchmark of Chapter 3 was used to measure the achievable Triad bandwidth. This ratio is pertinent when kernels are memory bandwidth bound; both the uplift in performance from using devices with improved memory bandwidth and the low number of FLOPs imply that the sweep kernel should be memory bandwidth bound. Indeed the computational intensity of the kernel is 0.22 FLOPs per byte of memory accessed: well within the memory bound portion of the Roofline models for most GPUs.

A perfect cache model is assumed, so that once a data point has been read it is available without further cost. There will be no data reuse between kernel invocations however, and so the model considers the movement for a single kernel. The number of reads and writes in a kernel, which corresponds to a particular wavefront, can be modelled as:

$$aN_{\text{cells}} + b \tag{5.3}$$

where a is the number of memory accesses which depend on the cell index and b is all other memory accesses. The number of cells in the wavefront is denoted N_{cells}. The values of a and b can be obtained by manually counting the read and write operations in the kernel source code and the specific problem sizes run (as listed in Section 5.2), for which the values $a = 61,400$ and $b = 866$ are obtained. For all kernels (wavefronts) the model predicts a total of about 15 GB of memory movement for the sweeps for a single iteration. The AMD CodeXL profiler for a run on the AMD S9150 GPU verifies that our model is fairly accurate, as it reported that 13.03 GB was moved for these kernels.

The sustained memory bandwidth may be calculated by dividing the modelled bandwidth by the runtime. For simplicity and consistency this same model is used for the original SNAP code running on the CPU. The sustained memory bandwidth is plotted as a ratio of Triad bandwidth in Figure 5.5. Note that the improved concurrent scheme achieves a similar percentage of peak memory

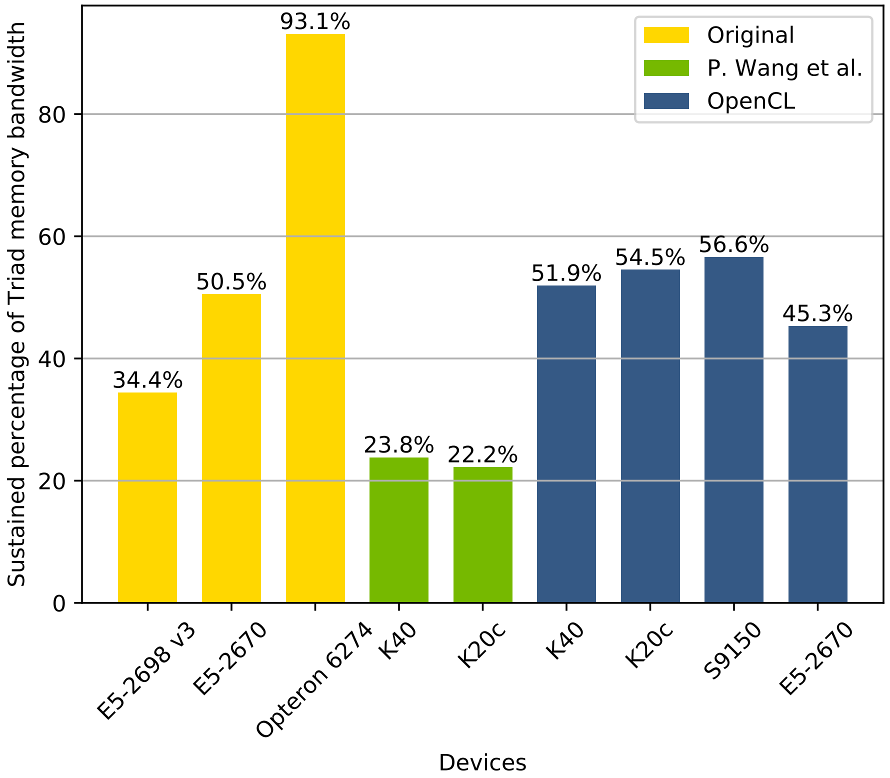


Figure 5.5: Sustained memory bandwidth of single node SNAP (from [24, 26])

bandwidth on GPUs to the original code on the Intel CPUs. This demonstrates that the advantage of more memory bandwidth on the GPUs is being leveraged successfully and therefore the speedups of Figure 5.4 are indeed in line with the memory bandwidth and the extra concurrency was required in order to provide the GPU with enough work so that the bandwidth may be leveraged. This is in contrast to the work of Wang et al. where the low percentages of Triad bandwidth in Figure 5.5 clearly indicate that memory bandwidth is not being leveraged efficiently. The AMD Opteron CPU achieves a very good percentage of achievable peak bandwidth, however notice from Table 5.1 that Triad is only capable of achieving 25% of theoretical peak memory bandwidth on these CPUs. Even the simple Triad operation does not leverage high levels of memory bandwidth on this platform.

5.4 Source code disruption

It should be noted that the improved concurrent scheme did require a number of source code changes. The memory layout of the main arrays needed to be changed so that the parallel scheme could use unit stride memory accesses. Therefore the arrays must be allocated and initialised in the new layout to avoid wasteful memory transpositions. As such this is a rather repetitive but straightforward update to the source as every location where the array is used must be changed. This results in a large number of simple changes. There is as yet no programming model which could assist with such changes (although RAJA may improve some aspects in this regard for C++).

The source update routines were simple to port to the GPU as every cell is independent and therefore more traditional HPC techniques were employed. However it was important that all the computation occurred on the GPU so that data transfer between the host and device was minimised.

The additional parallelism was required for the GPU in order to saturate it with sufficient work, something the original scheme is unable to provide. Integrating this within SNAP was somewhat troublesome due to way the parallel scheme has been expressed via manual work sharing between OpenMP threads. However the need for different parallel schemes informs us that a portable transport code should be designed with this flexibility in mind.

5.5 Mitigating memory capacity constraints

Standard CPU compute nodes may be constructed with large dynamic random-access memory (DRAM) capacity, however for accelerators the memory capacity is restricted. High bandwidth memory capacity on devices such as the NVIDIA P100 and Intel Xeon Phi (Knights Landing) (KNL) is currently only 16 GB. Future generations are likely to increase this to 32 GB but the rate of future growth is likely to be slow. The memory footprint of transport is very large and dominated by the angular flux solution. One option to mitigate the smaller memory capacity of accelerators is to strong scale the simulation to utilise a larger number of them. However, as will be discussed in Chapter 7, this is not an ideal solution. Storing more of the grid on each device reduces the number of MPI ranks which in turn should improve the scalability.

The use of the CPU host DRAM for capacity storage with device memory used for temporary working set storage is a commonplace solution in HPC, allowing access to the entire heterogeneous memory hierarchy [65, 31, 52]. Typically, the data is split into tiles which are then transferred to GPU memory by treating it as a programmer controlled scratch pad memory. For all the cited studies the order in which the tiles are computed does not matter, and the concurrency scheme treats each cell independently, and so these approaches cannot be applied directly to transport. Additionally, the methods hide the transfer of data between the host and device memory spaces by increasing the size of the halo region so that each tile may be resident on the device for a greater number of iterations. Again, this approach does not apply to a transport solver which does not utilise the standard halo exchange communication pattern. Kochurov and Golovashkin investigate applying similar techniques to a Gauss-Seidel solver which shares some characteristics with a transport sweep (see Section 4.6.2). They use a red-black parallel scheme whereby neighbouring cells are coloured differently and cells of the same colour may then be computed concurrently. Such a scheme breaks the upwind dependency of the transport sweep.

The NVIDIA CUDA programming model supports Unified Virtual Memory, allowing the device to use the host memory at its full capacity. However, there is no caching or prefetching of this data resulting in memory bandwidth limitations associated with the host to device interconnect (typically PCIe). This reduction in memory bandwidth is severe compared to what would be available by being resident in device memory.

The transport solver allows each octant to be solved in turn, and each octant sweep does not require any (cell centered) angular flux data from other octants. Therefore the working set of data is much less (an eighth of) than the total footprint. It is possible to manually program the transfer of angular flux data for each octant between the host and device in order to reduce the footprint on the device itself. As with the approaches to standard HPC codes discussed above, one hopes to overlap such transfers with compute. In order to achieve this, two octants may be stored on the device instead of all eight, with master copies of all eight stored in host memory. Therefore one octant may be computed simultaneously as the data for the previous octant sweep may be copied back and replaced with the next octant.

The PCIe interface supports duplex transfer so the interconnect indeed has support for such a scheme. However, when running on multiple nodes communications must occur and therefore data required by neighbouring ranks must be copied from the device. These transfers must occur at the same time as both the next portion of computation and the switch of octant data. The bulk transfers of the octant data therefore need interrupting to send the smaller neighbour messages. It was found that these transfers were only interrupted by both manually splitting the large transfer into multiple transfers *and* setting the stream priorities for the neighbour message copies to be higher than the bulk transfer stream. This resulted in some overlap, however there was still sufficient non-overlapped time to increase the runtime compared to a fully resident solution. GPU hardware and driver support for this will be crucial for large heterogeneous systems to be effectively leveraged.

An automatic page-faulting mechanism was introduced in CUDA 8 for Pascal and newer GPUs which would greatly simplify the implementation of this scheme. Indeed, one would only need to allocate the memory on the host and put faith in the memory manager to copy data as required. This is similar to the Unified Virtual Memory (UVM) available within the CUDA programming model, but with the addition of allowing data to be cached. However at the time of writing there was not support to test this.

A time dependent transport solve stores two copies of the angular flux, one for the current and previous timestep respectively. The values from the previous timestep are used to update the current timestep, yet it is the scalar flux for which convergence is checked in SNAP. The source updates do not depend on the angular flux either. Therefore it is possible to store a single copy of the angular flux array and perform an extra sweep once convergence has been achieved to overwrite the angular flux solution. This results in a 55% memory footprint saving and has been introduced into the official SNAP benchmark as an update after feedback to the developers. Note that this requires the scalar flux reduction operation to occur during the sweep itself which resulted in overheads on the GPU (see end of Section 5.1.2).

Using a discretisation method with higher orders of accuracy than second order diamond difference/finite difference (FD), such as a finite element method (FEM) approach, may also afford memory capacity savings. The higher order methods allow for using a reduced spatial resolution, consisting of fewer but physically larger cells. As such the memory footprint might be reduced as a result. This is the subject of Chapter 8.

5.6 Summary

The concurrent scheme presented in this chapter shows that the SNAP proxy application can be successfully run on a GPU to leverage the improvements in memory bandwidth that such devices provide over standard CPU architectures. Due to the low computational intensity of the sweep kernel where there are few FLOPs per byte of memory used, the kernel should be memory bandwidth bound. GPU devices offer an increased memory bandwidth capability but require large amounts of parallelism to be exposed in the algorithm in order to exploit it. Along with the concurrency exposed in the angular and energy domains as in the original SNAP scheme, spatial parallelism between cells was required to generate enough work to saturate all of the processing elements of a GPU. This scheme was tested experimentally and was the first time that speedups of SNAP were obtained on many-core architectures. The BabelStream benchmark of Chapter 3 was used to measure the maximum possible memory bandwidth, and along with a model of the memory movement in the kernel the attained memory bandwidth of the sweeps can be estimated. This verified that the speedups were in line with the improvements in memory bandwidth that GPUs offer over more traditional processors.

CHAPTER 6

Transport on cache-based architectures

The work in this chapter also appears in the following publications:

- Tom Deakin, John Pennycook, Andrew Mallinson, Wayne Gaudin and Simon McIntosh-Smith. *The MEGA-STREAM Benchmark on Intel Xeon Phi Processors (Knights Landing).* The Intel Xeon Phi Users Group Spring Meeting, 2017.
- Tom Deakin, Simon McIntosh-Smith and Wayne Gaudin. *On the Mitigation of Cache Hostile Memory Access Patterns on Many-core CPU Architectures.* The Intel Xeon Phi Users Group Workshop at International Conference on High Performance Computing, 2017.

After the release of the Intel Xeon Phi (Knights Landing) (KNL) there was much effort in the High Performance Computing (HPC) community to benchmark and port codes to this architecture. The device promised improved memory bandwidths over traditional x86 CPUs due to incorporating Multi-Channel DRAM (MCDRAM) into the package. For memory bandwidth bound applications therefore one would expect their performance to improve in line with this technology, and for some applications this was true [48]. However, and perhaps surprisingly, SNAP showed little improvement, achieving close to parity performance with standard Xeon processors [48]. SNAP however should be memory bandwidth bound based on the computational intensity as defined by the Roofline model (recall Section 5.3), and therefore on modern processors the limiting factor in the architecture should be the memory bandwidth. Recall too that the improvements shown in Chapter 5 were proportional to the memory bandwidth advantages of GPUs. It was unclear why the measured performance of SNAP was at odds with the expectations of improved performance from the KNL architecture. The focus of this chapter is therefore to understand why the improvements in memory bandwidth were not being exploited on cache-based architectures, of which the KNL processor is one such example, for a code with low computational intensity. Other cache-based architectures such as Intel Xeon

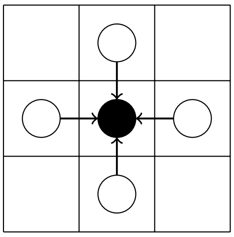


Figure 6.1: A standard 5-point stencil (from [22])

and IBM Power 8 processors are also used so that the findings may be described to this class of architectures and not just a single processor.

The mega-stream collection of mini-apps was therefore written based on extracting key sections of the sweep kernel in SNAP so that they may be optimised and their performance analysed in isolation. Although SNAP is a proxy application and therefore should allow for this style of research, it is unfortunately less agile than many of the other mini-apps (such as those found in the Mantevo suite [42]). This effort of distilling down the key components of the routine allows one to gain an insight into how the hardware may be limiting performance of the computational patterns in the code.

Critical to their development was the inclusion of a performance model which captured the *useful* memory bandwidth the application obtained.

6.1 Distillation of the finite difference kernel

The main computation involved in solving the transport equation results in a few lines of computational code organised in a deeply nested loop structure over the high-dimensionality of the problem space. This itself is then nested inside the iterative scheme of Figure 4.2, running the sweeps many times until convergence is reached. The new mega-stream mini-app seeks to take these few lines of code inside the loops over the problem space and optimise for memory bandwidth. This represents the core computation of each (SNAP) kernel invocation. Importantly, the memory access patterns and computational intensity are in line with those typical of a transport solve.

The kernel may also be described as a modification of a standard 5-point stencil operation in 2D (7-point in 3D). For a cell (i, j), this stencil relies on values from cells $(i \pm 1, j \pm 1)$ as shown in Figure 6.1.

For transport however edge values are used from neighbouring cells, rather than cell centred values. Also due to the upwind dependency, the origin of the sweep, only half the neighbour values are used for computation, with the others as output values to be consumed by neighbouring cells downwind of the sweep. For one particular sweep direction therefore, cell (i, j) requires edge values from cells $(i - 1, j)$ and $(i, j - 1)$ for computation, and then provides edge values for cells $(i + 1, j)$ and $(i, j + 1)$. This is shown in Figure 6.2.

In addition to this, for transport there are multiple values per cell from the angular (and energy) domains. It is in this angular dimension that compiler auto-vectorisation is applied (recall Section 5.1.1), and the mega-stream

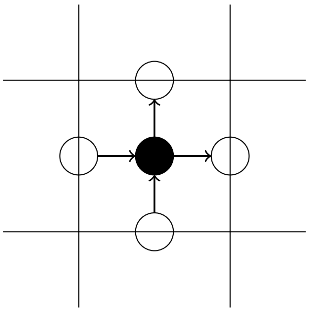


Figure 6.2: An upwind 5-point stencil (from [22])

mini-app also employs this strategy. Therefore the stencil operations occur on multiple data points per cell. This is in contrast to standard applications of the stencil as in Figure 6.1 where all of the parallelism is employed in the spatial dimension; vectorisation, threading (if used) and distributed domain decomposition all occur in space.

The mega-stream kernel (in Fortran) is shown in Listing 6.1. The calculation of the cell centred value on line 6 shares many characteristics of the STREAM Triad kernel; namely it consists of a number of multiplications and additions (which will hopefully result in fused multiply-add instructions being generated) and there is no reuse on the main input and output arrays. The ratio of floating point operations to memory *accesses* is close to 1 (resulting in a computational intensity of 0.125 with double precision). Calculation of the edge based values on lines 7–9 is again of low intensity and is just a simple finite difference (FD) relationship; a two point average (mean). Finally a reduction over all the cell centred data is performed as on line 10. The memory layout of each multi-dimensional array also mimics the sweep kernel in SNAP. There is stride one access to values in each cell corresponding to the multiple (angle) values in each cell, while the edge angular flux arrays are allocated on planes in the grid. This captures the majority of the essential elements present in the transport sweep kernel.

As the mega-stream kernel itself is representative of the work in a single chunk in SNAP, the parallelism is similar so as to be representative. An OpenMP work-sharing construct is added above the outer-most `m` loop shown in Listing 6.1, as if for running SNAP with a single Message Passing Interface (MPI) rank on a socket and using threads over the energy domain. The inner-most `i` loop is auto-vectorised by the compiler; this behaviour is ensured with OpenMP Single Instruction Multiple Data (SIMD) clauses. The extent of each loop may be set at runtime, with default values and the equivalent SNAP dimensions are shown in Table 6.1.

6.2 Optimisation of mega-stream

The mega-stream kernel shall be optimised focusing on maximising the (effective) use of memory bandwidth, as this should be the limiting factor for this kernel. The amount of memory moved by the kernel is modelled using similar assumptions to those of BabelStream in Chapter 3 and SNAP in Section 5.3.

```
DO m = 1, Nm
  DO l = 1, Nl
    DO k = 1, Nk
      DO j = 1, Nj
        DO i = 1, Ni
          r(i,j,k,l,m) = q(i,j,k,l,m) + a(i)*x(i,j,k,m) +
           ↪  b(i)*y(i,j,l,m) + c(i)*z(i,k,l,m)
          x(i,j,k,m) = 0.2*r(i,j,k,l,m) - x(i,j,k,m)
          y(i,j,l,m) = 0.2*r(i,j,k,l,m) - y(i,j,l,m)
          z(i,k,l,m) = 0.2*r(i,j,k,l,m) - z(i,k,l,m)
          total(j,k,l,m) = total(j,k,l,m) + r(i,j,k,l,m)
        END DO
      END DO
    END DO
  END DO
END DO
```

Listing 6.1: The mega-stream kernel (from [22])

Loop	SNAP equivalent	Range
Ni	nang	128
Nj	nx	16
Nk	ny	16
Nl	nz	16
Nm	ng	64

Table 6.1: Default mega-stream loop extents

Specifically, memory movement is counted by examining the kernel source by hand and assuming that once a data value is read it remains in cache and does not need to be read (and therefore counted again) from main memory. The model therefore captures the read of the read-only arrays (q, a, b and c), the write of the write-only array (r) and the update consisting of a read and a write of the remaining arrays (x, y, z and total). By dividing the amount of memory moved by the runtime of the benchmark, the *estimated* memory bandwidth can be known. It is this bandwidth that captures the useful and necessary memory movement; any other movement of data is wasteful from an algorithmic perspective. By optimising for the improvement of this modelled bandwidth the runtime must decrease.

The optimisations employed focus on cache effects and behaviour, and notably ensure that:

- data which *is not* re-used *is not* in cache
- data which *is* re-used *remains* in cache
- data is in cache in time for use

These factors reduce cache pollution and ensure the temporal locality of data in cache, thereby reducing the latency for data to arrive from high levels of the memory hierarchy. Note that the memory access pattern is not especially changed with these optimisations for it is already vectorised code with a predictable, stride one access pattern; factors which constitute a ‘good’ access pattern. However, these optimisations do bring about significant performance improvements.

6.2.1 Reducing cache pollution

The `q` and `r` arrays representing the cell-centred angular flux arrays, are read- and write-only respectively. There is also no re-use of the data in these arrays for the entire kernel; behaviour such as this is known as streaming. Therefore there will be little benefit from the data being in the cache, unless the arrays are small enough to remain resident for the entire duration of the application. However these are large arrays and their footprint exceeds the capacity of even the last level cache. For the write-only `r` array specifically it is therefore better that it is not in cache at all.

On Intel architectures, the caches are write back, and therefore for a standard memory store the data must first be read into (L1) cache before it can be updated, a policy known as “read for ownership”. For `r` however this data is write-only and therefore first reading the data is wasteful. Through the use of compiler directives it can be ensured that a non-temporal store instruction is issued so that the data may be written directly to main memory thus removing the redundant read. As a result the data is no longer expected to be in cache leaving more capacity for other arrays.

6.2.2 Ensuring cache residency

The `x`, `y` and `z` arrays representing the edge based angular flux arrays have an interesting reuse pattern. They are allocated on planes of the spatial mesh and so therefore each have one spatial dimension missing. This means that in the loops over space (`j`, `k` and `l`) the data in these arrays is reused. For example the same entry of `z` is used for all values of the `j` loop, although the value in this entry is updated for each iteration of `j` in turn. Therefore, it would be hoped that this data remains in cache for the duration of the loops taking advantage of temporal locality. A similar pattern can be observed for the `x` and `y` arrays.

Although these arrays do not contain the full spatial domain and individually may be resident in cache, their combined footprint often renders them too large to fit in low levels of the cache hierarchy. For example the size of each of these arrays for the default input sizes as shown in Table 6.1 is 16 MiB. If this problem was run on a KNL processor using 64 OpenMP threads, each thread would own 256 KiB of each array, with a total of 768 KiB for all three arrays per thread. Each tile in the processor has 1 MB of L2 cache shared between two cores, and so assuming there are no collisions each core has 512 KB of L2 cache available, which is smaller than the size of the arrays. On this processor, due to the lack of an L3 cache, when these arrays fall out of L2 cache they must be read from main MCDRAM memory, imposing a significant latency penalty (recall Figure 3.2 on page 22).

There are no compiler directives, intrinsics or other ways to explicitly control what data should remain in cache. Therefore a cache blocking, or tiling, approach in order to reduce the size of the working set is one of the limited options that are available to indirectly control cache residency. The inner-most (i) loop is therefore blocked in groups of eight entries; each cache line is 64 B and so blocking by eight corresponds to a single cache line. This does however introduce a requirement that Ni is a multiple of eight. Note that tiling is generally employed in the spatial dimensions to increase the spatial locality of data in the cache, but for mega-stream tiling is used to improve the temporal locality and is not performed in the spatial dimension. For the example problem by tiling on a cache line only 16 KiB of each of the edge flux arrays is needed in cache at any point in time, reducing the total working set of these arrays to only 48 KiB which is well within the capacity of the L2 cache on KNL.

Implementing tiling however is a rather intrusive modification. An extra loop is inserted into the already deeply nested loop structure, which is deeper still in SNAP due to the iterations on the source. Also the data allocation and all accesses of the array must be modified to match the new data layout, and therefore this change must be propagated throughout the entire code base.

6.2.3 Ensuring data is in cache in time

Most modern HPC processors contain hardware and software prefetchers, allowing data to be moved from main memory and stored in cache before this data is requested by a load instruction. The hardware prefetchers operate without user intervention and try to predict the memory access pattern for each stream of data. If user intervention occurs for a stream of data through the use of software prefetch instructions the hardware prefetcher ceases to operate for that particular stream. The compiler may try to insert prefetch instructions if the appropriate optimisation flags are enabled, and the user may insert intrinsic instructions manually specifying the prefetch distance. Selecting an appropriate prefetch distance may involve much trial and error.

By profiling the mega-stream mini-app with the tiling and non-temporal store optimisations applied it was found that there were a high number of L2 cache misses for the q array when running on the KNL. By adding in software prefetch intrinsic functions with a distance of 32 vector instructions, the next part of the array can be moved into cache ahead of when it is needed, resulting in it being in cache in time for use thus reducing the number of cache misses.

The optimised kernel is shown in Listing 6.2.

6.2.4 Results

The effect these optimisations have on the runtime, and therefore the modelled memory bandwidth is investigated on different cache-based processors. The KNL processor has two levels of cache, with L2 shared between pairs of cores situated on a tile. Main memory consists of high bandwidth MCDRAM, as well as double data rate dynamic random-access memory (DDR) which is unused here. Two generations of Intel Xeon processors, Broadwell and Skylake, are also tested. Both Xeon processors have three levels of cache, with only L3 shared between cores. The L3 cache in Broadwell is inclusive and so contains a copy of all L2 data, whereas on Skylake L3 cache is exclusive. Recall from Figure 3.7 that the KNL should provide the most memory bandwidth overall, with a Triad result of 448 GB/s, with dual-socket Xeon processors Skylake achieving 191 GB/s and Broadwell 130 GB/s.

```
DO m = 1, Nm
  DO n = 1, Ni/8
    DO l = 1, Nl
      DO k = 1, Nk
        DO j = 1, Nj
          CALL MM_PREFETCH(q(1+32*8,j,k,l,n,m), 1)
          !DIR$ VECTOR NONTEMPORAL(r)
          DO i = 1, 8
            r(i,j,k,l,n,m) = q(i,j,k,l,n,m) + a(i,h)*x(i,j,k,n,m)
            ↪  + b(i,n)*y(i,j,l,n,m) + c(i,n)*z(i,k,l,n,m)
            x(i,j,k,n,m) = 0.2*r(i,j,k,l,n,m) - x(i,j,k,n,m)
            y(i,j,l,n,m) = 0.2*r(i,j,k,l,n,m) - y(i,j,l,n,m)
            z(i,k,l,n,m) = 0.2*r(i,j,k,l,n,m) - z(i,k,l,n,m)
            total(j,k,l,m) = total(j,k,l,m) + r(i,j,k,l,n,m)
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

Listing 6.2: The optimised mega-stream kernel (from [22])

Processor	Architecture
Xeon Phi 7210	Knights Landing
Xeon E5-2699 v4 (22 core)	Broadwell
Xeon Gold 6152 (22 core)	Skylake
Power 8 (10 core)	POWER

Table 6.2: List of devices used for the mega-stream experiment

The modelled memory bandwidth achieved by mega-stream on each architecture is shown in Figure 6.3, with optimisations implied inclusively. Details of the hardware are shown in Table 6.2. The default problem size was run, with one OpenMP thread per physical core. On the Xeon processors therefore some threads may have more work than others to do, resulting in an imbalance. However for a fair test a single problem size was used across architectures.

The baseline performance of mega-stream on these architectures is poor. It is particularly striking that the Xeon Phi has similar performance to Xeon despite having much more memory bandwidth available, a key motivator for this work, thus showing that mega-stream is somewhat representative of SNAP in this regard. The mega-stream kernel has a low computational intensity and should be memory bandwidth bound, but these results suggest otherwise.

The issuance of non-temporal store instructions makes a marked difference on the KNL architecture, with the performance improving by 3X. Note however that the memory bandwidth is still low compared to a Triad kernel. A smaller improvement of 1.3–1.5X is seen on Xeon architectures. The effect of the cache

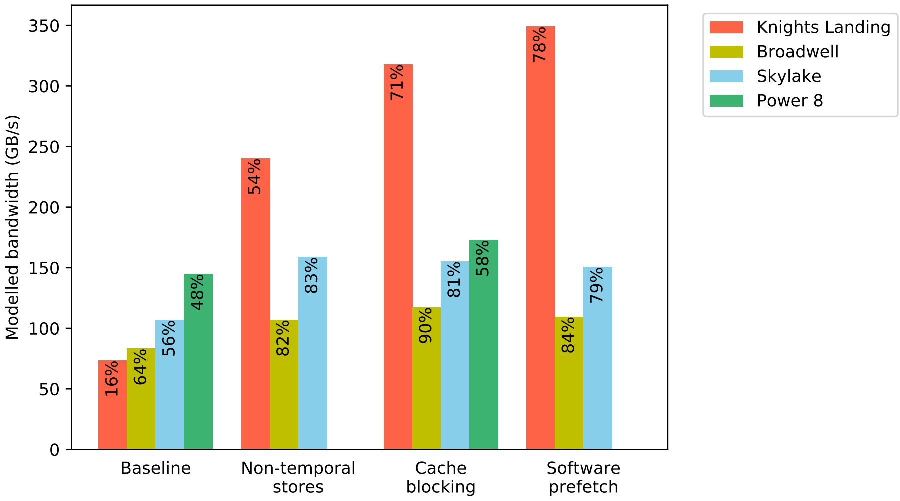


Figure 6.3: Modelled memory bandwidth for mega-stream mini-app for default problem (from [22]) with labels showing percentage of measured Triad performance

pollution caused by the r array is therefore most significant on KNL, which has limited cache capacity.

Ensuring cache residency by tiling all the arrays again improves the performance. As before this is most noticeable on KNL which improves by a further 1.3X. This optimisation reduces the size of the working set, and therefore increases the number of cache hits in all levels of the cache hierarchy. For KNL, if data is not available in L2 cache, the memory latency to read from the next level of the memory hierarchy is huge as it is MCDRAM. On Xeon, falling out of L2 results in reading from the large L3 cache, where the latency is still much better than main memory, however still occurs a penalty.

Adding software prefetching instructions to ensure the q array is in L2 cache in time for use helps the KNL architecture only by 10%. For the default problem size on both Xeon processors this step actually reduces the performance. The hardware prefetchers on Xeon architectures are more sophisticated than on Xeon Phi, and so when an imbalanced workload is run on this architecture they do a better job than a fixed programmer controlled prefetch value; indeed they may alter the prefetch distance as the algorithm progresses and as cores with less work become idle. Note on KNL the default problem tested resulted in a balanced workload per thread. For problem sizes where each thread has the same amount of work on the Xeon CPUs a small improvement in performance is found.

Overall therefore these optimisations improve the performance of the mega-stream kernel so that it achieves over 80% of Triad bandwidth on KNL, resulting in a 4X speedup from the original baseline performance. On Xeon processors the speedup is approximately 1.5X, again an improvement of note although not as significant as on KNL. The deeper and larger cache hierarchy of Xeon therefore mitigates the need for these optimisations, as well as masking the necessity for optimisation as the baseline was not performing poorly enough to cause concern on Xeon; only on Xeon Phi was this noticeable, however the optimisations helped both architectures.

The Power 8 is a cache-based processor from IBM which unlike the Intel processors does not use the x86 instruction set. The mega-stream mini-app was run on this processor too, with one thread per physical core, with the baseline performance measured at 145 GB/s. Whilst this is greater than even the optimised version of the code on Intel Xeon architectures, the Power 8 achieves 299 GB/s memory bandwidth on the BabelStream/STREAM Triad kernel (which is also quite low compared to the theoretical peak of 384 GB/s — this may be down to the simultaneous multithreading (SMT) modes in this processor restricting the functional units available to each thread). Therefore even higher performance should be expected. Unfortunately, the Power instruction set does not contain a non-temporal store instruction, so it is not possible to prevent cache pollution with this technique [44]. Note that as seen in Figure 6.3, the majority of the performance improvements from the three optimisations came from the use of non-temporal store instructions. The cache blocking optimisation does improve the result to 173 GB/s, however this is still far short of Triad performance.

6.3 Porting mega-stream optimisations back into SNAP

There was much scope for optimisation of the mega-stream mini-app despite already having well vectorised code with stride one memory access patterns; the key criteria for exploiting memory bandwidth. The optimisations presented allow the memory bandwidth of cache-based architectures to be leveraged more successfully and result in improvements to the runtime. Reimplementing these suggested optimisations back into SNAP itself however poses a challenge.

Firstly, SNAP utilises Fortran 90 array operations meaning there is not a single loop over the angular dimension. Although the arrays do all iterate over the same bounds, the compiler is unable to merge them due to the branching caused by global options. Despite rewriting the kernel to use an explicit loop rather than array notation, it remained these branches which prevented the generation of the streaming store instructions — this is thought to be due to the number of different combinations of branch directions to multiversion. These branches are global in the sense that they determine options based on the input settings, and importantly all lanes in the vector will go the same way. It is likely the number of options which prevents the compiler multi-versioning the routine as there would be a large number of combinations of options.

Additionally, the write for which a streaming store would be beneficial does not occur every iteration as is modelled in mega-stream. As a memory footprint saving optimisation, the angular flux is not updated until the source terms have converged, and then a final sweep occurs overwriting the angular flux in place, thereby removing the need to retain two copies of the angular flux array (as described in Section 5.5). However, this means that the write does not occur as frequently and the benefits from streaming stores will not be as great a benefit; note that it was the streaming stores which provided the most benefit overall. It is of note that although the compiler report confirmed that with the appropriate compiler directives and branch and loop refactoring a streaming store was generated, the runtime of the application did not change significantly, as it had with mega-stream.

Implementing the tiling approach into SNAP would require a significant rewrite of the majority of the code base. Array slices are used to pass sections of the arrays corresponding to different energy groups through a number of subroutines. Combined with the Single Program Multiple Data (SPMD) approach to OpenMP parallelism used in SNAP, it would be very invasive to tile the arrays in its current form. If a new transport application was to be written however, it would be advised not to use array notation and not to pass slices through subroutine interfaces as well as adopting a more conventional approach to OpenMP parallelism so that these optimisations may be applied in a more straightforward manner.

As such, the optimisations of mega-stream were unable to be applied to the parent proxy application, SNAP. It is important to consider that as a mini-app, mega-stream removes some complexities of SNAP in order to focus on specific issues. Therefore further work is required to include more complexity into mega-stream in order to capture more of the behaviour of SNAP itself, so that it is possible to determine to precise aspects of the hardware which are inhibiting improved performance. This motivates the proceeding section of this chapter on the development of the mega-sweep mini-app.

The mega-stream mini-app does not model the compute in a way that is comparable to the GPU scheme presented in Chapter 5. This benchmark considers the computation of a single chunk of the spatial domain and is parallelised in the same way as the original SNAP benchmark as described in Section 5.1.1. As such a GPU implementation of mega-stream would not be representative of the GPU port of SNAP. However the GPU port of SNAP is limited by the memory bandwidth as shown by the model in Section 5.3, as expected for such a code with low computational intensity, and the potential for further optimisations may not be as significant as it was for the KNL architecture.

6.4 Introducing extra complexity to mega-stream

The mega-stream mini-app captures the loop structure and computation required in the very centre of a transport solver. In order to better capture the behaviour of SNAP (as the mega-stream optimisations did not significantly benefit the parent application as detailed in Section 6.3) more complexity was added to mega-stream to form the new mega-sweep mini-app.

The central solve of mega-sweep contains the same elements as mega-stream, with the addition of the denominator term which introduces a divide operation. Also, as sweeps occur from all corners of the mesh, the concept of octants was introduced so that the data is accessed according to the forward and backward iterations; this tends to have little effect on performance but was introduced for completeness. Importantly this allows both for different communication patterns for each octant sweep and ensures the memory footprint of the angular flux array is representative.

The spatial decomposition using the Koch, Baker and Alcouffe (KBA) scheme was also added and implemented using MPI routines so as to capture the communication patterns (details of the KBA algorithm are introduced in Section 7.1); specifically after each invocation of the most central (mega-stream style) kernel, messages must be sent.

Thread based parallelism via OpenMP was still included over the energy domain as in mega-stream.

With these inclusions, the only major component of the SNAP sweep kernel that is not included is anisotropic scattering. However even without this feature the mega-sweep mini-app is already showing a performance degradation compared to mega-stream.

The similarity between the mega-stream and mega-sweep kernels can be seen by comparing Listing 6.1 with Listing 6.3. The calculation of the loop bounds has been suppressed in Listing 6.3 for clarity. For example, the computation of the angular flux `psi` has a similar numerator to the mega-stream calculation of `r`. Also the outgoing edge flux calculations `psii` and `psij` are identical to the updates to `x`, `y` and `z`, in particular with respect to the looping structure and data reuse.

```
DO sweep = 1, nsweeps
  DO c = cmin, cmax, jstep
    CALL recv(psii)
    DO g = 1, ng
      DO cj = ymin, ymax, jstep
        j = (c-1)*chunk + cj
        DO i = xmin, xmax, istep
          DO a = 1, nang
            psi = (mu(a)*psii(a,cj,g) + eta(a)*psij(a,i,g) +
             ↪  v*aflux0(a,i,j,sweep,g)) / (0.07 + 2.0*mu(a)/dx +
             ↪  2.0*eta(a)/dy + v)

            psii(a,cj,g) = 2.0*psi - psii(a,cj,g)
            psij(a,i,g) = 2.0*psi - psij(a,i,g)
            aflux1(a,i,j,sweep,g) = 2.0*psi -
             ↪  aflux0(a,i,j,sweep,g)

            sflux(i,j,g) = sflux(i,j,g) + psi*w(a)
          END DO
        END DO
      END DO
    END DO
  CALL send(psii)
  END DO
END DO
```

Listing 6.3: The mega-sweep kernel

The mega-sweep mini-app again demonstrates the philosophy of distilling just the key components of an algorithm into a small application so that performance issues may be captured. The resulting kernel is simple enough that a performance model may be constructed, an essential part of this methodology. The performance model counts the number of read and writes to main memory in a cache oblivious manner, as per mega-stream, and by dividing this by the runtime of the program a model of main memory bandwidth utilisation is formed.

The mega-stream mini-app, on which mega-sweep is based, was able to achieve high levels of memory bandwidth as shown in Section 6.2.4, yet the mega-sweep mini-app is unable to achieve the same. The use of non-temporal store compiler directives to ensure correct generation of non-temporal writes to the angular flux array was implemented as per the optimisations to mega-sweep. For the mega-stream mini-app, this optimisation provided the majority of the improvements (as in Figure 6.3); on Skylake (Xeon Gold 6152, 22 cores), adding non-temporal stores improved the mega-stream bandwidth from 56% to 83% utilisation, a change of 27 percentage points. Although this optimisation does also improve the performance of the mega-sweep mini-app when the number of angles (the inner most loop) is a multiple of the vector width, the memory bandwidth is unable to be leveraged to the same extent. For mega-sweep, non-temporal stores improved the modelled bandwidth from 48% (91.7 GB/s) to 65% (123.3 GB/s) of main memory bandwidth utilisation, a change of 17 percentage points. Note too that the utilisation is much lower for mega-sweep than for mega-stream even with this optimisation in place. For a number of angles which is not a multiple of the vector width, as for common choices of Discrete Ordinates (S_n) quadrature set, the performance is much worse due to the inability to use non-temporal stores due to lack of memory address alignment. A noticeable lack of effect of using non-temporal stores is observed with the SNAP proxy application too where the quadrature set is not necessarily a multiple of the vector width.

The communication also contributes to much of the runtime, even for calculations running just on a single node. On the Skylake processors, running with one MPI rank per core (totalling 44 MPI ranks) 27.5% of the runtime was in communication, as measured on just the master rank. As such this does include the start up and tear down time associated with the different sweep directions. This is a necessary part of the algorithm however, and therefore it is important to capture the cost of this.

When running on a single node, the total amount of work is constant irrespective of whether flat MPI is used for parallelism or OpenMP threads are used to parallelise over the energy groups. Using OpenMP alone should also remove the start up and tear down costs associated with the sweep, with the other communication costs associated with running just a single MPI rank being negligible. However, this is much slower than using MPI alone; a symptom also noticed in the SNAP benchmark. Compared to the 123.3 GB/s (with a runtime of 35 s) for flat MPI, using OpenMP threads alone only achieved 105 GB/s (with a runtime of 41 s). As the communication costs are negligible in the later case, the computation time is the total runtime, whereas in the flat MPI case, the compute time was 25.6 s excluding communications. Therefore, although the total work performed in both cases is the same, choosing to apply the hardware resource to different problem dimensions (group or space) results in different performance profiles.

Importantly, the shape of the outgoing neighbour flux arrays (`psii` and `psij`) is different depending on the decomposition which itself is a consequence of the application of the parallelism. Due to the reuse of the data in these arrays, the best performance should be attained when these arrays are resident in cache. For example, when parallelising over energy groups (flat OpenMP), each core has a slice of the arrays that are of the size of the *full* spatial mesh times the number of angles. Whereas, when parallelising over space (flat MPI), each core has a slice of the array that is a *portion* of the spatial mesh times the number of angles; the number of energy groups is ignored as they are operated on serially in turn and appropriate hardware prefetching such ensure cache residency. Therefore, the footprint of these arrays *per core* is likely larger when running flat OpenMP than flat MPI and therefore may indeed generate additional cache pressure.

As such, the current working hypothesis for the performance limiting factor of the mega-sweep mini-app, and therefore the SNAP benchmark due to the similarity in performance profiles, is in fact due to cache memory accesses, rather than main memory bandwidth. This is unintuitive due to the requirement to stream a very large array, the angular flux, which has no reuse within the kernel and is far too large to fit in any level of the cache. Typically such requirements lead the kernel to become bound by the memory bandwidth. Further work is required to investigate how the properties of the cache hierarchy may effect the performance of mega-sweep. This might include the use of hardware counters to measure cache miss rates which could provide evidence for determining which arrays are sensitive to cache level. A preliminary study into how the problem size effects the performance shows the typical cliff edge drops in performance as array sizes are increased, although it has not yet been possible to associate these with the cache level or particular arrays.

6.5 Summary

Two new mini-apps were developed and presented in this chapter. Their aim was to capture the important performance characteristics of the main solve kernel in a deterministic S_n transport application. Firstly, the mega-stream mini-app captured the main computation in the kernel. Optimisations focussing on ensuring cache residency of the appropriate arrays demonstrated a 4X speedup over the baseline implementation on KNL, with smaller improvements on Xeon processors. The issuance of non-temporal store instructions for the store of the angular flux array gave the most marked improvements. However, on applying the mega-stream optimisations to the SNAP proxy application similar improvements were not found. The mega-sweep mini-app was therefore written to investigate these disparities. Early results from this show that whilst non-temporal stores do improve the performance of mega-sweep, they do not result in so large a speedup as with mega-stream. Further work is required using the mega-sweep mini-app in order to confirm that the performance limiting factor of a transport code is the caches rather than main memory bandwidth.

The mini-apps have been made available online at `https://github.com/uk-mac/mega-stream`. The benchmarks have been written without the need for an input file, with all problem dimensions set on the command line. The majority of the results in the chapter have been generated using the default problem size. Version `0.3` of mega-stream was used, marked with a DOI of `10.5281/zenodo.1203611`. Formal versioning of mega-sweep has yet to be employed due to its current active development; the git commit used throughout this chapter is marked `758047b`. This has been marked with a DOI of `10.5281/zenodo.1203614`.

Both mini-apps leverage performance models which describe the memory bandwidth. These models are built into the mini-apps and therefore useful metrics about the performance can be calculated directly from the runtime of the application. Such models are important in investigating the performance of an application as runtime alone is not sufficient to determine the efficiency to which hardware is utilised.

CHAPTER 7

Scalability of transport

The work in this chapter also appears in the following publications:

- Tom Deakin, Simon McIntosh-Smith and Wayne Gaudin. *Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale.* International Conference on High Performance Computing, 2016.

Spatial decomposition involves taking a mesh of cells and allocating parts of the mesh to different processing elements. These processing elements are usually represented as Message Passing Interface (MPI) processes. As such each process performs the computation on a portion of the mesh, known as a sub-domain. There are a variety of ways in which this decomposition can occur and each has an effect on the scalability of the computation. However all must orchestrate the computation so that the data dependency of the upwind sweep is adhered to whilst allowing each process to work in parallel as much as possible.

Additionally, due to the large memory footprint of the angular flux itself due to its high dimensionality, no one computational node is able to fit the entire solution in memory. Therefore decomposing is essential just to fit the problem in memory. Although large problems in most High Performance Computing (HPC) applications rarely fit on single node, the issue of being memory capacity bound is more acute for transport.

This chapter will show that although scaling the problem over many computational nodes is required in order to provide sufficient memory capacity, the scaling properties of the transport solver are far from ideal. The concurrency scheme for GPUs will also be tested at scale and it will be shown that despite limited capacity and scalability of the decomposition scheme, the concurrent scheme of Chapter 5 is still a viable approach for the current and future scale of supercomputing systems.

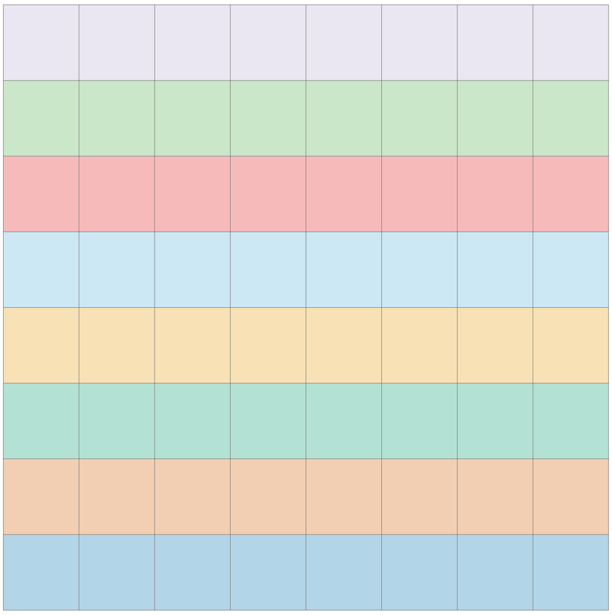


Figure 7.1: Illustration of KBA decomposition of a 2D mesh

7.1 The Koch-Baker-Alcouffe decomposition

The Koch, Baker and Alcouffe (KBA) algorithm or decomposition is a spatial domain decomposition and associated sweep schedule that is prevalent in transport codes [50, 12, 11]. The decomposition of the spatial domain is performed in one less than the dimensionality of the problem; for a 2D problem the domain is split across a 1D list of processors, and for a 3D problem the domain is split across a 2D grid of processors. As such each sub-domain consists of the full extent of one of the spatial dimensions, along with the angular and energy group domains. Due to their long, thin shape the sub-domains are typically called *pencils*. An example KBA decomposition for a 2D mesh is shown in Figure 7.1 and for a 3D mesh in Figure 7.2; in both figures each sub-domain is represented by a colour.

The intuition behind the decomposition is that the initial processor, starting with the corner starting cell of the sweep can continue work after completing enough work for the neighbouring processor to begin work. Additionally, work should continue for as long as possible to reduce the time that the processor lies idle. Traditionally, the cells are swept on the first processor until a boundary is reached, at which point the outgoing angular flux is sent to the neighbouring processor which requires this boundary as part of the upwind dependency. This second processor can then begin work on its sub-domain whilst the first processor continues along the pencil. The time taken between the first processor starting work and for the final processor to begin work is the *start up* time. Processors lie idle during this start up time and so the parallel efficiency is reduced. As such one can never expect perfect scaling.

Note that this communication pattern is different to the prevalent halo-exchange found in many common HPC codes, where all the processors can

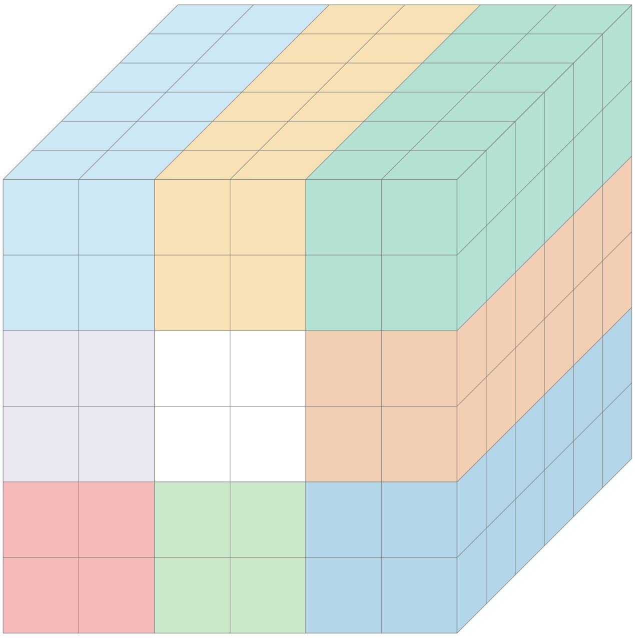


Figure 7.2: Illustration of KBA decomposition of a 3D mesh (from [25])

begin the computation and the boundary cells on the sub-domains are swapped simultaneously. In a transport code the processors can only begin once their boundary dependency is met via upwinding, and the communication follows a 'push' pattern. The processors wait in a blocking `MPI_Recv` call until the neighbour data is sent. This is an unavoidable effect of the spatial sweep dependence. The synchronisation between nearest neighbours using a blocking receive call gives sufficient coordination between processors to ensure the sweep dependency is correct whilst minimising wider reaching synchronisations.

It is common to describe the sweep dependency across the spatial mesh in terms of a directed acyclic graph (DAG) such as that in Figure 7.3 adapted from [40]. Here a 4 × 4 2D structured spatial mesh is decomposed between 2 processors, with the upwind dependency for one quadrant (equivalent to an octant in 3D) shown as arrows between cells. The figure also highlights the start up time of Rank 2, which cannot begin until Cell 2 has been computed and its outgoing value communicated. As Rank 2 begins computing Cell 3, Rank 1 can continue computing starting with Cells 9 and 6, with the latter's outgoing value again needed by Rank 2.

Once the sweep for a particular angle has been completed, the data dependency for all other angles in the same octant have this same dependency graph, and so the processors can begin the next angle in the octant and the processors do not remain idle. This is only true in the structured grid case; in an unstructured grid each angle has its own (possibly unique) dependency graph (DAG). This is because as the Discrete Ordinates (S_n) angle is cast through the spatial mesh, following the neighbour dependencies to produce the graph, the cells in the unstructured mesh may be traversed in a different order depending on the chosen angle. Once all the angles in the octant have completed however, the

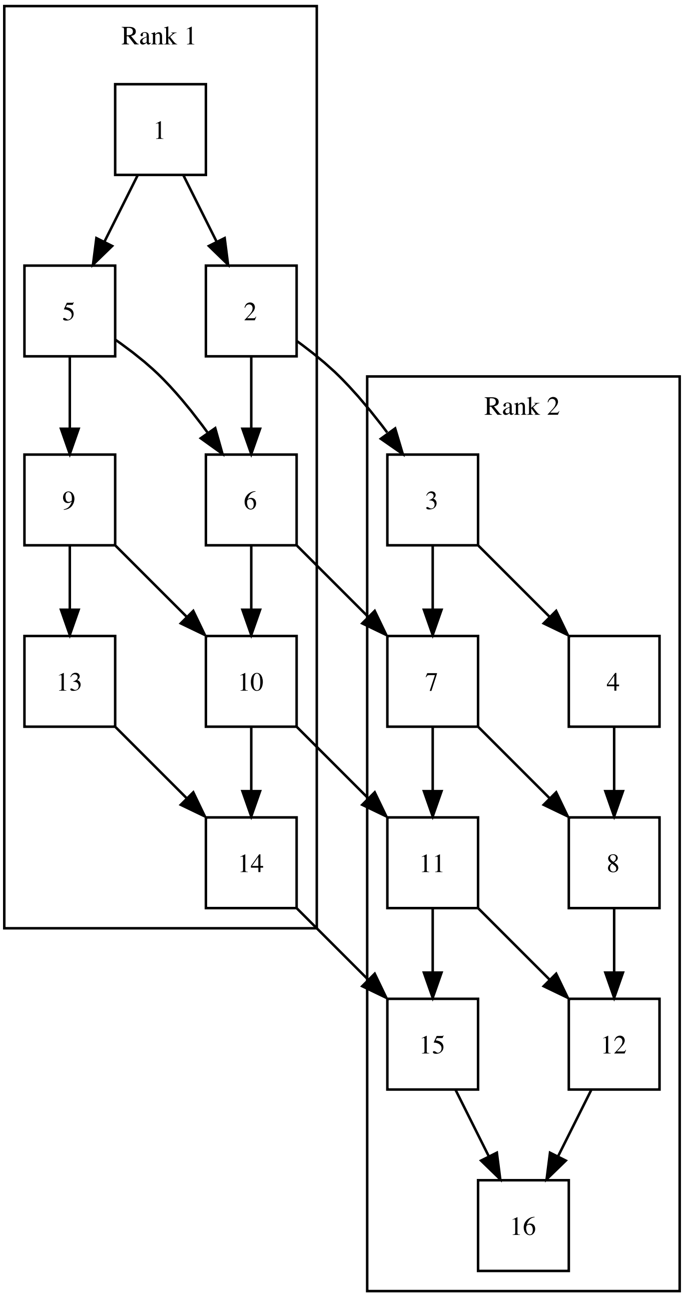


Figure 7.3: A DAG for a KBA sweep adapted from [40]

processor has finished its work and must wait until the next sweep direction from a new octant is started. This propagates across the grid, the processors becoming idle in symmetry to them becoming active at the start up of the sweep. This could be said to be a *tear down* time. A new DAG must begin for the new sweep in the next octant.

The choice of octant determines whether the initialising processor must wait or can continue useful work preventing it from becoming idle. If the decomposition is in the YZ plane, then the sub-domains have the complete extent of X. If the next octant chosen represents the opposite x stride to that coming before, then the processor can begin work immediately, beginning the sweep from the other boundary corner in the sub-domain. After completion of the first sweep, the boundary conditions will be met for this new starting cell, either from the vacuum boundary conditions or the new completed solution of the reflective boundary condition. As such two octants are said to be *pipelined*, and completed in order and this removes a start up and tear down cost for each octant pair. At the end of each octant pair there must still be a period of tear down.

There is a sweep for every angle and energy group and the KBA algorithm has been extended by Baker to consider more modern CPU architectures where each core has vector units [13], where all angles in an octant are computed simultaneously using the vector units, and thus outgoing fluxes for all angles are communicated to neighbouring ranks in a single message. Note that although the messages are larger than a single double precision floating point number, they are still small enough to not be limited by the network bandwidth and as such may be more affected by the network latency given current interconnect performance. The energy domain was also treated independently via the Jacobi scheme (recall Section 4.2.3) so that threads could be launched to enable sweeping each group concurrently. Baker showed that this had little difference at small node counts, but at higher node counts it was a much more scalable approach. It is this approach that is also utilised in SNAP, whereby all angles within an octant for a single energy group are computed using the vector units. This approach was originally noted by the authors of KBA who found that higher performance was achieved by at least computing all angles within a single octant [11].

A modification of the KBA algorithm, known as *hybrid-KBA* was proposed by Adams et al. [2]. In this scheme each process has half of the spatial domain which is usually complete in standard KBA. For a 2D grid therefore, the mesh is decomposed across a mesh of $p \times 2$ processors. This further improves the scalability, again only with regard to the maximum number of processors which the problem may be run on. However, where reflective boundary conditions occur this extra decomposition cannot occur and the hybrid scheme reduces to the standard KBA algorithm.

7.2 Other decomposition schemes

The KBA schedule is the *de facto* standard decomposition for transport codes, with all other decomposition schemes comparing their performance to KBA. However it is relevant to survey some of these other schemes, and in particular focus on their suitability when running on many-core architectures.

Many schemes were written with the IBM Blue Gene/Q supercomputer in mind, primarily the *Sequoia* machine installed at Lawrence Livermore National Laboratory (LLNL). The Blue Gene architecture consisted of a very large number of nodes with good nearest-neighbour communication properties including low cost synchronisation. Additionally the cores were low power embedded cores with around 1 GB of dynamic random-access memory (DRAM) per core, much less than other large machines available at the time. This is in comparison to the more usual 2–3 GB per core (or higher) found in other systems [98].

Applications on Blue Gene were therefore required to scale to high numbers of cores. For transport applications utilising the KBA decomposition, this posed a challenge due to limits of spatial decomposition. For example, what would seem a very large 2D mesh of 4000 × 4000 cells would be able to run on a *maximum* of 4,000 cores; Sequoia has 1.5 million cores, and so 4,000 is a very small fraction of this machine. Therefore other decomposition schemes were investigated in order to improve the scalability of transport. Note that 'scalability' in this context primarily refers to the ability to scale further, rather than having close to perfect scaling (as defined in Section 2.2).

A common focus of the alternative schemes is over-decomposition of the spatial domain so that more than one block of cells is allocated to each process. The KBA algorithm already contains over-decomposition of the spatial domain from decomposing in fewer dimensions than the spatial dimensionality. Specifically the first processor continues processing cells along the pencil domain after the first communication so that this process has more than one block of cells in which to compute. This would not be the case if a standard decomposition was performed typical in halo-exchange codes where the processors would sit idle after computing their sub-domain as the sweep continued.

For the alternative schemes in general, the sweep schedule for the total work is broken up and aggregated into smaller blocks or tiles in a task based manner. Therefore (in this more general scheme) as the sweep progresses a process may have to make a choice as to which block of cells to compute before another, assuming both have their boundary conditions satisfied. An initial study by Bailey and Falgout describe the point at which a choice must be made as 'collisions' in the sweep schedule [9]. They used three different schemes to prioritise the computation of tiles when collisions occur and showed that over-decomposing did improve the scalability of transport sweeps using their algorithm. In addition they showed that KBA does weak scale well up to 10,000 processors, as did Hoisie et al. who predicted that the KBA algorithm would be dominated by the computation rather than the communication; Section 7.4.1 will show otherwise. however the latter authors' analysis was on a theoretical 100 TFLOP/s system consisting of 20,000 processors [43].

Adams et al. proposed a scheme which was optimal in the sense that the sweep occurs in the minimum number of stages given the decomposition and aggregation of work [2]. In their scheme the sweep schedule consisted of angular sets, group sets and cell sets, and the scheduler was designed to compute these in an order so as to minimise the number of stages required. They also presented a performance model for their optimal sweep schedule that was able to predict the runtime given the aggregation factors (such as the number of angles in the angle set), along with an optimisation routine to select these parameters to ensure good scalability. Optimality was also proved for the KBA schedule along with their 3D over-decomposition scheme, however their performance results where restricted to hybrid-KBA decompositions.

The work on porting SNAP to GPUs from Chapter 5 of this thesis shows that good performance was obtained by computing all angles and groups concurrently; therefore there is one angle set and one group set. The scalability properties of the schedules proposed by Adams et al. rely on there being multiple angle and group sets, and indeed the proofs of optimality use a single angle per angle set. This is not feasible when considering good GPU performance of the solve itself where all angles and groups were used in a single set. Therefore the length of the task pipeline becomes significantly reduced, which hampers the scalability of the scheme. Therefore there is a tension when selecting the aggregation factors between the scaling properties of the schedule and the appropriate parameters for different architectures.

An improvement to the overloaded scheme of Adams et al. was proposed which uses non-contiguous domain overloading [3]. Unlike the other schemes discussed the spatial domain assigned to each process need not be contiguous. The schedule was presented as a modification of the previous schedule via overloading and demonstrated improved scalability given the right aggregation factors. However the scalability in particular relies on having a large number of angle sets. On modern CPU architectures vectorising the angular dimension was shown to give good performance [13], which will again result in a single angle set. By not exploiting this vectorisation scheme, it is again unlikely that this new scheme will actually give good performance in practice.

Overall these alternative schedules do not take a holistic approach where the time taken to solve each chunk of work may not be linear in the size of that chunk. For example, on a GPU starved of sufficient work, the computation may take just as long as a situation where sufficient work is available. Again the models also assume that switching between different chunks of work has low overhead; an approach which does not necessarily guarantee good memory access patterns either.

7.3 Modelling sweep algorithms

There have been a number of attempts to model the performance of sweep schedules in order to assess the efficacy of the various approaches. These models attempt to capture how effective a schedule is for a given problem size running on some number of parallel processors. The focus has been primarily on CPUs, however as a contribution to this thesis Section 7.3.3 will extend one such model to be valid across both CPU and GPU devices.

7.3.1 Parallel computational efficiency

The Parallel Computational Efficiency (PCE) is a simple ratio of the amount of useful work required compared to the amount of work performed in total [50, 11, 13]. The amount of useful work required is simply the number of angular flux calculations — the product of all problem dimensions. The amount of work performed is defined as the product of the number of stages in the schedule and the maximum amount of work per schedule. The number of stages in each sweep is the minimum number of levels in the DAG required to traverse it. For the DAG in Figure 7.3 for the 4×4 grid the number of stages in the sweep is seven, and corresponds to the horizontal levels in the DAG. The number of stages is also the number of wavefronts across the mesh.

Note that this ratio does not consider any overheads or communication costs and as such the PCE gives a crude metric as to how the sweep schedule performs. The original definition also did not include in the model the number of parallel processors. Koch et al. noticed that in the KBA schedule which swept one angle at a time the PCE was >75% for a variety of grid sizes yet performed slower in practice than the alternative KBA schedule which swept all angles within the octant which has a PCE of 30–60% [50]. The second method has lower communication latency costs due to fewer messages but this is not captured by the model.

Adams et al. provide a formula for the PCE which includes terms for the processor counts [2], and was used in Baker's later work [13]. This update captures the relationship between the size of the mesh as well as the communication pattern, for it is the processor array that determines the communication pattern whereas the mesh size determines the computational load per processor between the messages.

7.3.2 LogGP based models

There have been a number of models exploring the communication costs of wavefront sweeps [75, 92] based on the LogGP modelling framework [4]. Pennycook et al. modify one such model to take into account computation costs on a GPU [79]. However, these models are overly complex compared to the other models presented in the chapter, consisting of a great many parameters. As such they are not sufficiently flexible to be transferable to general schedules as they are specific to particular incarnations of schedules.

7.3.3 A time aware model

A model for the time taken for the sweep algorithm by Bailey and Falgout has been proposed [9] based on a similar model proposed earlier by Mathis et al. [64]. The model is defined simply as the number of stages in the sweep algorithm multiplied by the time taken for each stage. The time for each stage is the sum of the computation and communication time, which includes the latency which depends on the number of messages. This model therefore provides an improvement upon PCE by including communication costs, but does not contain any measurement on whether the number of stages is efficient. However the PCE gives sufficient measure of this.

The model was originally expressed with CPU parameters. As part of this thesis the model has been enhanced so that it may also be applied to GPU architectures [25]. The model predicts the runtime of the sweeps T for a 3D spatial problem according to the following rules:

$$T = S(C + B + L) \tag{7.1}$$

$$B = \beta m \Gamma \tag{7.2}$$

$$L = \alpha K \tag{7.3}$$

$$S = 4\left(P_x + P_y - 2 + \frac{2N_z}{\eta}\right) \tag{7.4}$$

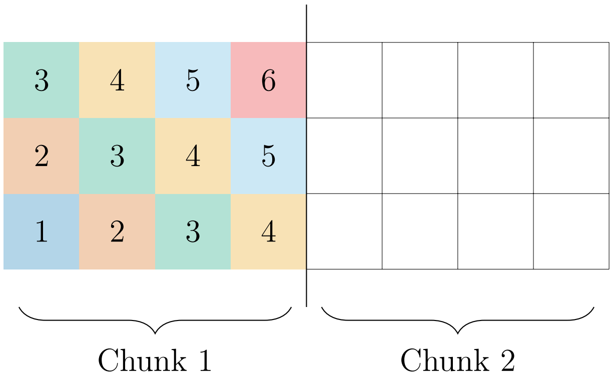


Figure 7.4: Illustration of chunking in KBA (from [25])

The interconnect latency L is modelled in (7.3) simply as the number of messages K times the time to send a message α. The interconnect bandwidth B is again modelled in (7.2) simply as the product of inverse network bandwidth β (the time to send a byte) and the size of the message. For ease the message size is expressed as the number of cells in the message Γ times the number of bytes per cell m. The machine dependent parameters α and β may be obtained via standard interconnect benchmarking tools.

The number of stages (7.4) is derived from the KBA schedule forming a 2D processor grid of P_x by P_y processors. Each stage corresponds to the computation of a chunk of η XY-planes on the sub-domain before each communication; the communication therefore occurs every chunk rather than as soon as a boundary is reached, an optimisation employed in SNAP to increase the size of messages and reduce the message count.

An example chunk of size $\eta = 4$ is illustrated in Figure 7.4 for a 2D grid. Communication to neighbouring processors occurs once the entirety of Chunk 1 has completed, whence boundary conditions for Chunk 2 are received and computation for it may begin.

Note that when using the wavefront parallel scheme in Section 5.1.2, the number of wavefronts utilising the maximum width of the pencil is reduced due to a start up and tear down for each chunk. Using a larger chunk size means more cells may be computed concurrently with this scheme, however communication is delayed and the start up cost between MPI ranks may increase. For the original parallelisation scheme described in Section 5.1.1 a larger chunk size results in serialisation between MPI ranks.

The number of stages described by (7.4) is equivalent to the number of levels in the DAG if the graph nodes are chunks of cells rather than individual cells.

The computational time C was originally modelled by Bailey and Falgout for CPUs as:

$$C_{\text{CPU}} = \gamma N_m N_g \eta \frac{N_x}{P_x} \frac{N_y}{P_y} \tag{7.5}$$

This is constructed as a ‘grind time’ γ to compute one entry of the angular flux multiplied by the number of updates in each stage: the product of the number of cells in the chunk and the angle and energy dimensions. The grind time γ gives a regression factor to allow tuning of the model.

As a contribution of this thesis a GPU computation time is added to the model. The intuition is that a kernel under-utilising the GPU, which occurs at the start and end of the sweeps, will have a similar runtime to running a kernel with sufficient work to saturate the GPU; this is as a result of the inherent overheads for offloading a kernel. The work per stage is therefore defined in terms of the number of wavefronts (corresponding to the number of kernel enqueues) rather than the size of the work per cell as in the CPU model.

$$C_{\text{GPU}} = \gamma \left(\frac{N_x}{P_x} + \frac{N_x}{P_x} + \eta - 2 \right) \tag{7.6}$$

Again γ is included as a regression parameter and is used to estimate the compute cost of each kernel.

The sum of these quantities $C + B + L$ gives the computational time per stage, and so the total runtime is simply this quantity multiplied by the number of stages as shown in (7.1).

This model has been validated at scale, as shown in [25] and Section 7.4.

7.3.4 Parallel sweep efficiency

The PCE and original time aware models were combined by Adams et al. to produce the parallel sweep efficiency model [2]. This was defined as:

$$\epsilon = \frac{T_{\text{task}} \times N_{\text{tasks}}}{N_{\text{stages}} \times (T_{\text{task}} + T_{\text{comm}})} \tag{7.7}$$

where T represents time and N represents the number (or count) as further denoted by the subscript. As the number of stages in the sweep algorithm is the number of tasks plus the number of idle stages, (7.7) may be rearranged as:

$$\epsilon = \frac{1}{\left(1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}\right)\left(1 + \frac{T_{\text{comm}}}{T_{\text{task}}}\right)} \tag{7.8}$$

The inclusion of timings in (7.8) allows machine dependent values to be included so that how a sweep algorithm might perform on different hardware can be directly estimated. These timings may be obtained by simple benchmark applications as is usual for standard communication times, or else derived from existing runs of transport codes based on the measured grind time for a single angular flux update. The grind time is the time taken by a transport code to update an angular flux value (see Glossary).

7.4 Accelerating transport at extreme scale

The computation of solving the transport equation has been accelerated through the effective use of GPUs as discussed in Chapter 5. This work continues by investigating how scalable this approach is combined with the standard KBA schedule on large supercomputers. The GPU accelerated port of the SNAP mini-app is therefore run on the two largest GPU-enabled supercomputers, Piz Daint and Titan. At the time of experimentation, both these machines contained NVIDIA K20X GPUs, but differed primarily in the interconnect and CPU technologies (see Table 2.1 on page 8).

7.4.1 Weak scaling

The weak scalability of the application is tested by running a pencil shaped sub-domain on each GPU. Note that each sub-domain would be pencil shaped if a large problem was decomposed, and therefore weak scaling in this way is representative. Therefore the total amount of work per GPU remains constant as the number of nodes is increased. The following problem was used:

- $4 \times 4 \times 400$ cells per MPI rank
- 136 angles per octant
- 32 energy groups
- 1 timestep of 0.01s
- Convergence criteria of 1.0E-5
- 2 orders of anisotropic moment scattering
- Communication chunk size of 4

This problem required 3.6 GB of storage for the large angular flux array, and therefore is within the 6 GB capacity of the K20X GPU.

One GPU is assigned to each MPI rank, and the scaling study will begin at four MPI ranks. As a comparison, the original SNAP code is run on the CPUs with the same problem size per MPI rank. On CPUs the code was run with 2 MPI ranks per non-uniform memory access (NUMA) region and sufficient OpenMP threads to ensure all cores were utilised; a configuration which was found to perform well on both system's CPU architectures.

From Figure 7.5 showing the runtime on Titan, the GPU implementation with the improved parallel scheme is leveraging a 4X speedup over the CPUs on this machine, both at small and large scale. It should be noted that perfect scaling, which would appear as a horizontal line on this graph, should not be expected; the performance model itself does not suggest that perfect scaling is possible. However the application does scale up to many thousands of ranks. The CPU runtime does degrade at large scale but this is likely due to known issues with network performance when running at large scale on the Gemini interconnect on Titan [33].

Similar good scaling can be seen in Figure 7.6 on Piz Daint. Here the improved GPU scheme achieves 2X over the original code running on the CPUs. The CPUs on Piz Daint are expected to be more performant than those in Titan, with the CPUs alone delivering around a 2X speedup for this code and problem. Therefore the GPU is delivering the other 2X performance improvement. Again this improvement is maintained at scale.

It it noticeable that the Aries interconnect on Piz Daint does provide better scalability than the Gemini interconnect on Titan at higher node counts, which can be deduced from the fact that the measured runtimes are much closer to what the model predicts. Additionally the runtimes on Piz Daint are faster than those on Titan, even though both machines contain the same GPU accelerators. Therefore the Aries interconnect is showing a clear advantage over the Gemini interconnect for this application, irrespective of the computational hardware.

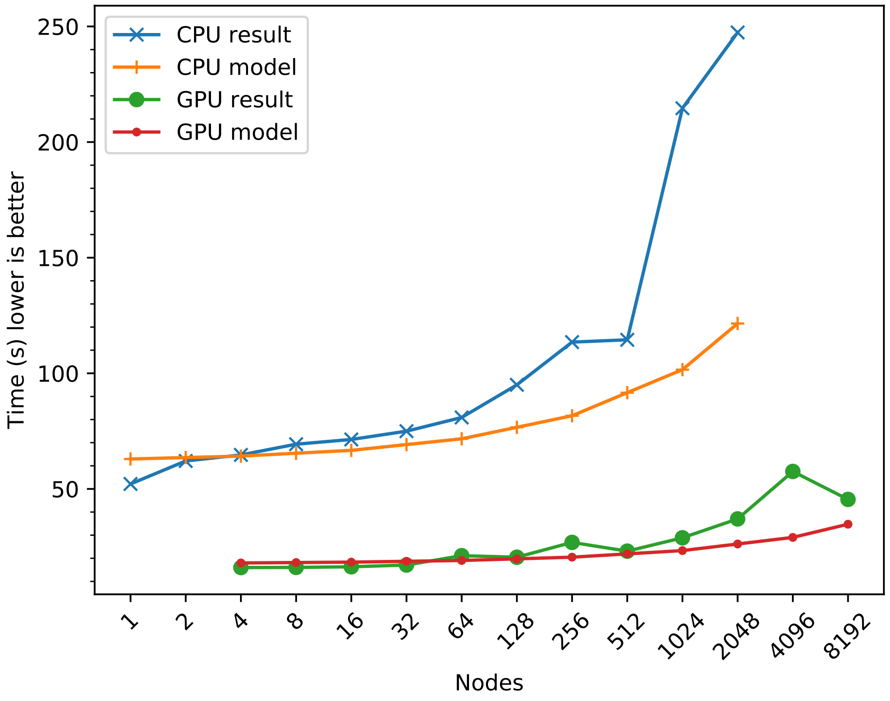


Figure 7.5: Weak scaling SNAP on Titan (from [25])

The results from the `b_eff` benchmark[1] on Titan and Piz Daint highlight the practical differences between the interconnects beyond the differing topologies. The Aries interconnect demonstrates improved bandwidth of 6354 MB/s/node and latency of 1.735 µs over the Gemini interconnect with bandwidth of 575 MB/s/node and 3.327 µs [25]; Aries is showing an 11X improvement in bandwidth and a 2X improvement in latency. As these experimental results were taken during the day-to-day operation of the machine, they represent the realistic performance of the interconnect on production systems running a usual load. The Gemini network also has different network performance properties depending on the direction travelled through the torus, and so the non-smooth performance of the GPU runs in Figure 7.5 may be as a result of rank placement on the machine. Therefore the improved routing and topology of Aries to avoid the congestion noticed by (for example) Freed et al. on the Gemini network will also contribute to the measured performance of these metrics beyond the improvements at the hardware level [33].

On both machines however the time spent in communication increases as the number of nodes increase. This trend is common in many HPC applications and is usually the cause of degradations in scalability. In this case for the GPU code running on 2,048 nodes, 80% of the runtime on Titan and 60% on Piz Daint is in communication alone; with the remainder in compute which in fact takes a constant runtime regardless of scale, as expected due to a fixed problem size per MPI rank under the weak scaling experiment. This contradicts the balance

[1] `https://fs.hlrs.de/projects/par/mpi/b_eff/`

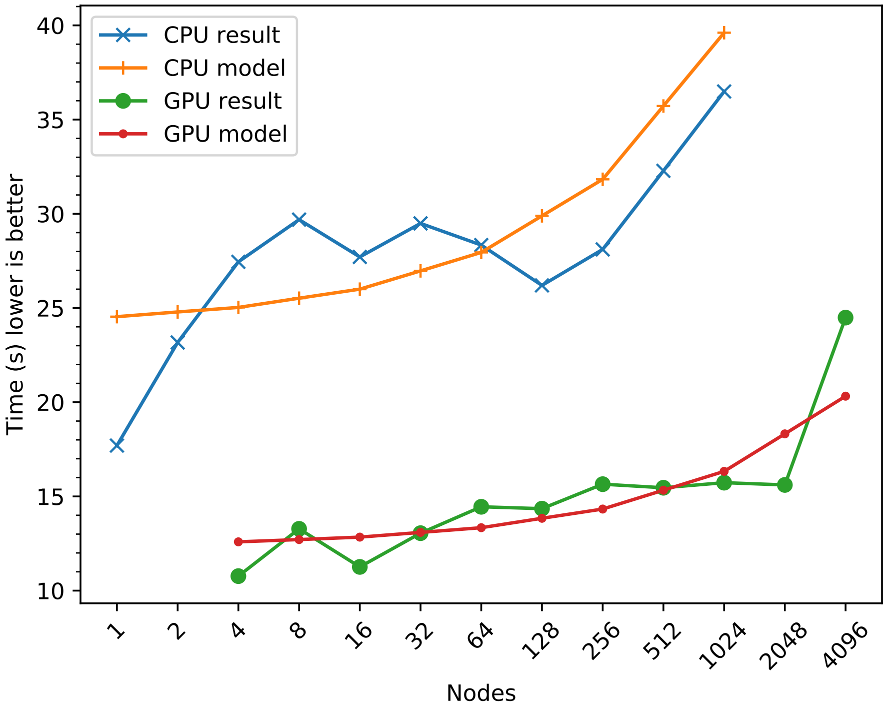


Figure 7.6: Weak scaling SNAP on Piz Daint (from [25])

of communication and computation predicted by Hoisie et al. [43] where it was predicted that computation would dominate at this scale. The trends observed by McCalpin also suggest system balances different to those predicted by Hoisie et al. (recall Section 2.1).

The network performance therefore is hugely critical to the performance of this algorithm at scale. Reducing the runtime of the computation through the use of GPU accelerators has highlighted a key requirement for improved network performance. Note that the messages will not be fully utilising (and therefore will not be limited by) the available interconnect bandwidth as they are not very large in size, and it is therefore the latency, along with the ability to fully asynchronously send messages, that is important.

This scaling study does show that KBA does indeed scale well enough for GPU accelerated codes, even up to large node counts. Near-future supercomputers which contain GPU accelerators are unlikely to reach the node count of Titan, with machines such as Summit and Sierra looking at around 5,000 nodes; this is similar in size to Piz Daint. Indeed, Piz Daint was recently upgraded to use newer NVIDIA P100 GPUs and has overtaken Titan in the Top 500 listing, achieving ranking 3 [91]. Therefore the KBA schedule remains a viable solution for solving the transport equation on these pre-Exascale accelerated machines.

7.4.2 Strong scaling

It is also important to consider the ability to decrease the time taken for an application to run by increasing the compute resources. By using more processors one hopes that the runtime of the particular input problem decreases. Ideally one hopes too for perfect strong scaling, with the runtime reducing linearly with the number of processors. Transport however does not weak scale perfectly and so therefore it is unlikely to strong scale perfectly either, as will be shown.

Strong scaling a deterministic transport application is a challenging endeavour. Strong scaling studies begin with a large enough problem so that it can be sufficiently decomposed to a large number of processors. Such a large problem will demand a large memory footprint and therefore finding a sufficiently large problem to decompose on a large number of processors yet fit within the memory capacity of just a few nodes is challenging. There are few strong scale results presented in the transport literature for this reason.

It is the spatial dimension which is decomposed primarily across processors (MPI ranks) and therefore a large mesh size is chosen. The angular and energy domains are chosen so that the memory footprint is not prohibitively large. Therefore the following problem was run for which the angular flux solution has a memory footprint of 690 GB:

- $256 \times 256 \times 256$ cells
- 10 angles per octant (S_8)
- 32 energy groups

The other choices of input which determine the iteration count are the same as for the weak scaling study of Section 7.4.1.

Both CPU and GPU runs of SNAP were performed on Titan and are shown in Figure 7.7. The CPU runs were flat MPI and did not utilise OpenMP threads

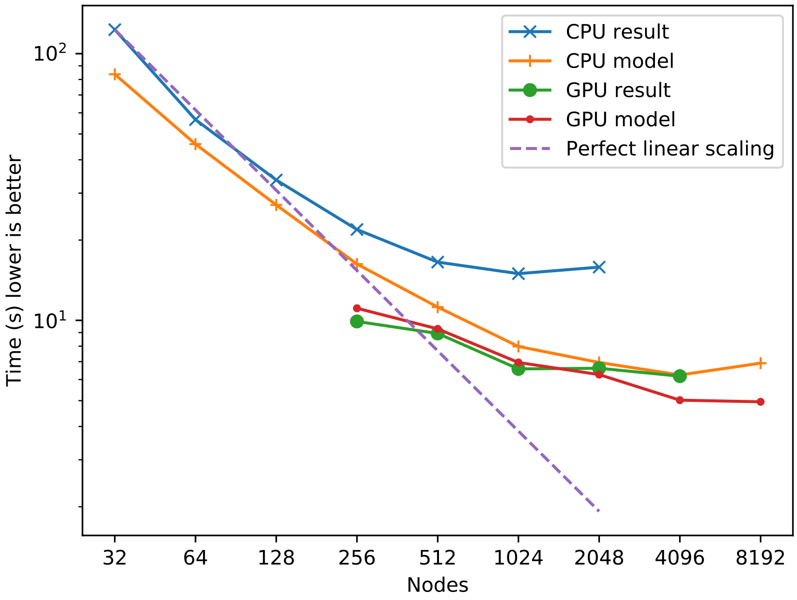


Figure 7.7: Strong scaling SNAP on Titan

due to segmentation faults occurring; a problem inherent in the current version of the SNAP proxy application. The model for predicting the runtime of the application as described in Section 7.3.3 was also applied here and shown in Figure 7.7. Of note is that perfect scaling is not predicted by the model; indeed the scaling is quite different but nevertheless the model and obtained runtimes match well.

Due to the limited memory capacity of GPUs it was not possible to run this problem on fewer than 256 GPUs on Titan; the additional memory capacity of GPUs newer than those on Titan will aid significantly in running transport applications. In particular the strong scaling of the KBA algorithm is modelled to be close to linear at the start, and therefore much can be gained by running on slightly more resource than is strictly necessary, but less improvement will be observed at significantly greater scales.

7.5 Summary

The KBA schedule is a stalwart of many transport solvers, including the SNAP proxy application used throughout this thesis. The schedule describes the decomposition of the spatial domain across distributed processors along with the sweep schedule, to ensure the wavefront dependency is respected whilst reducing both the synchronisation between processors and processor idle time.

Other more recent schedules have been surveyed and it is found that their properties for good scalability are incongruent with the GPU concurrency scheme of Chapter 5. Additionally these schedules are always compared to KBA and many do not provide an advantage over Baker's most recent descriptions of KBA [13].

The GPU implementation of the SNAP proxy application utilising the improved concurrent scheme was run at scale on the two largest GPU enabled supercomputers. The concurrent scheme was shown to still perform well at scale, for both weak and strong scaling situations, even on small sub-domain sizes. As such the scheme does allow GPU accelerators to be leveraged to provide runtime improvements for the solution of deterministic transport.

Such scaling was verified using a performance model which captures both the computation and communication properties of the transport sweep. Through extending the simple time aware model of Bailey and Falgout to allow for modelling the computation in SNAP on both CPU and GPU architectures, the runtime of the proxy application running at scale on both architectures can be verified. The model also confirms that perfect linear scaling of a transport sweep should not be expected, and indeed the experimental results also demonstrate this behaviour. In addition this shows that the KBA scheme is still good enough to provide sufficient scaling on supercomputers designed around advanced computer architectures.

The source code for the GPU implementation of SNAP used in the chapter, along with the problem input files used to run on the Titan and Piz Daint supercomputers, are available online at `https://github.com/UoB-HPC/SNAP_MPI_OpenCL`, with a DOI of `10.5281/zenodo.1203633`.

CHAPTER 8

High order finite element solution

The order of accuracy for a particular discretisation method is given in terms of the cell width h. An n^{th}-order accurate method has error $O(h^n)$. The finite difference (FD) discretisation used so far in this thesis is second-order accurate and so has error $O(h^2)$. A higher order method generally allows for a larger cell width to maintain a similar error to a lower order method.

Transport applications are often memory capacity bound and so the low memory capacity of high bandwidth memories such as Multi-Channel DRAM (MCDRAM) and High Bandwidth Memory (HBM) pose a problem. This chapter will explore the linear discontinuous Galerkin (DG) finite element method (FEM) with an aim to show that sufficiently coarse cells may be used to allow for a memory footprint saving.

The FEM is more often associated with unstructured meshes as it allows generation of a mesh with different cell sizes, shapes and orientation. However the focus in this chapter will be on using it on a structured mesh, although the mathematics will be identical to an unstructured mesh; it is just the mesh connectivity which is different. As such, the work in this chapter provides some exploratory findings which could form the basis of future study as detailed in Section 9.1.

The SNAP proxy application from Los Alamos National Laboratory (LANL) will be extended in order to compare the original FD implementation and the FEM implementation practically.

The reader is referred to Appendix B for the important concepts of the DG-FEM method and their application to the transport equation itself.

8.1 Comparison to the finite difference discretisation

As can be seen from the descriptions of the FD discretisation in Section 4.2.1, the FD method is relatively simple. In contrast, the FEM is rather more complex (as shown in Appendix B). Both methods involve inversion of the streaming-collision operator, although the inversion is more explicit with the FD method; the FEM approach does invert the operator but it is done during the solve of the small linear system in each element. The FD method uses the simple diamond difference relations of (4.3) to calculate outgoing fluxes at cell faces. The FEM does not generate outgoing fluxes explicitly; rather nodes from neighbouring elements are used directly.

The number of floating point operations to evaluate each of the diamond difference relations in the FD method is just a single multiply-add operation; one for each spatial dimension. On the other hand, the FEM requires many more floating point operations, in particular for the solve of the small linear system. The `dgesv` routine from Linear Algebra PACKage (LAPACK) requires $0.67N^3$ operations for N nodes; in 3D where $N = 8$ this is over 300 FLOPs just for the solve of one of the linear systems. The assembly of the linear system for each of the dimensions in the angular flux requires additional floating point operations. Therefore much more work is required to solve the equation using FEM compared to FD for each point in the domain.

The computational intensity of both methods is important in terms of assessing how they perform relative to each other. In Section 5.3 it was stated that the FD solution in SNAP has a computational intensity of 0.22 FLOPs per byte. For the FEM implementation with linear elements described in Section 8.2, where the element and boundary integrals of pairs of basis functions are precomputed, the computational intensity of the FEM kernel is 0.25 FLOPs per byte. The amount of memory moved during matrix construction biases the solution to again be bound by memory movement rather than FLOPs despite the much more numerically intensive solve. With higher order elements (linear were used to obtain the quoted values) the matrix solve may begin to dominate.

The FEM stores a solution of each unknown in the angular flux at each node (for linear elements this is on each vertex); recall that the unknown dimensions are space, angular direction and energy group. For FD only a single value is stored per unknown, with neighbour fluxes stored in temporary arrays which are not significant to the memory footprint. The memory overhead for a 3D linear FEM mesh is therefore eight times that of a FD mesh, for a fixed mesh size. However the FEM offers a higher-order accuracy solution than a FD approach. The linear DG elements are third-order accurate, whereas the FD approach is second-order accurate. Therefore in practice (for a given error) the FEM allows the use of physically larger cells, and thus coarser grids consisting of fewer cells may be used to provide a suitable solution. As such the increase in memory footprint resulting from storing multiple nodes for each unknown might be mitigated through the use of a coarser grid. This relationship is modelled in Section 8.3.2.

8.2 Implementation details

The SNAP benchmark used throughout this thesis uses the FD method for spatial discretisation, and is used as a baseline for comparisons with a new FEM port. In order to test the viability of the FEM, the SNAP benchmark is converted to use the method. In this way, parallel implementations of both methods are made available in order to practically evaluate them at scale.

Unlike many other approaches utilising the FEM, the spatial grid used is regular and structured, rather than unstructured. This is so that the additional challenges of an unstructured transport sweep are not introduced (see Section 9.1) and a direct comparison between the methods can be made available. Although the indirect memory accesses of an unstructured mesh are not present in this implementation, the method has been applied assuming the cells may be arbitrary hexahedra; in particular the Jacobian is calculated for each cell as it would be in an unstructured grid, even though this may be simplified for a structured mesh. This will allow for further expansion to an unstructured grid as part of a future study.

Recall from Section 5.1.1 that the original SNAP benchmark implements the FD method using Message Passing Interface (MPI) to decompose the spatial domain according to a Koch, Baker and Alcouffe (KBA) schedule (as described in Section 7.1), and uses OpenMP threads to parallelise energy groups and Single Instruction Multiple Data (SIMD) instructions to parallelise over angles within the octant.

The new FEM port of SNAP also uses the KBA schedule for spatial decomposition. The assembly and solution of the small linear system in each cell introduces an additional level of potential concurrency. For the implementation this step uses SIMD instructions auto-generated by the compiler. As with original SNAP, OpenMP threads are used to parallelise computation over energy groups. The iteration over the angles within the octant is conducted serially, taking each angle in turn, with the octants pipelined as governed by the KBA schedule. The angle loop could be collapsed with the energy group loop so that OpenMP threads may be used for both domains, but this has not been tested, because if an extended version to an unstructured mesh was implemented extra angular dependencies are introduced negating the concurrency in this domain.

The FD scheme calculates and subsequently communicates the outgoing angular flux data. The FEM scheme instead directly communicates the angular flux data for the appropriate face nodes.

Note that a limitation of the original version of SNAP is that it does not allow for uneven mesh decompositions. The FEM port on the other hand does allow the mesh to be divided across any number of MPI ranks for convenience.

8.2.1 Solving the linear systems

The linear hexahedral elements in this implementation corresponds to the construction of an 8-by-8 matrix with an 8-wide vector for the right hand side of the linear system $A\psi = b$. This small system then needs to be solved to find the unknown ψ.

Two schemes were tested for solving this system; firstly using the Intel Math Kernel Library (MKL) implementation of the LAPACK `dgesv` routine to solve the system, and secondly a hand-written direct Gaussian elimination and backwards substitution routine. The matrix would be classed as a 'small matrix', however small matrix libraries such as LIBXSMM only contain matrix-matrix multiplication routines (`dgemm`) so cannot be used [41].

As the sweep progresses across the mesh, a matrix is assembled in each cell, for each angle and energy group. Due to the parallel scheme used, a matrix is created for each cell and angle in turn, with multiple matrices existing in parallel depending on the number of OpenMP threads used to parallelise the energy domain. It may be possible to use batched matrix routines, generating and storing multiple matrices and passing to the Application Programming Interface (API) for parallel processing. To maximise the parallelism, matrices for all angles and groups in a single cell could be assembled, stored and solved in this way. This may allow for more reuse of the integrated test functions which would be shared by all matrices in the cell batch. The on-node parallelism could then be organised by the math library itself, rather than dictated by the programmer. Note that many routines in math libraries are not optimised for small matrices, and optimisations for small matrices usually occur for batched routines [1, 93]. For linear elements using S_{32} and 32 energy groups, where for a structured grid they may all be solved in parallel, storage of 2.2 MiB per cell (for the matrices) and solving a batch of 4,352 systems. On a GPU with parallelism exposed on cells on each wavefront, the memory footprint will be more significant due to the need to exploit spatial concurrency. As this study forms a precursor to unstructured mesh sweeps, it may not be possible to use a batched routine on an unstructured mesh as the concurrency in the angular domain is reduced.

The MKL routine `dgesv` uses lower upper (LU) factorisation to split the matrix $A = L \times U$ where L and U are lower and upper triangular matrices respectively. Triangular matrices are simple to invert, so that it is easy to calculate the solution ψ using forward and backward substitution:

$$A\psi = b \tag{8.1}$$

$$A = L \times U \tag{8.2}$$

$$c = L^{-1}b \tag{8.3}$$

$$\psi = U^{-1}c \tag{8.4}$$

However the process is inherently sequential, processing one row at a time.

The ease of inverting an upper triangular matrix is illustrated in Figure 8.1 for a 4-by-4 matrix, and the process is similar for a lower triangular matrix. This forms a series of simultaneous equations:

$$u_{1,1}\psi_1 + u_{2,1}\psi_2 + u_{3,1}\psi_3 + u_{4,1}\psi_4 = b_1 \tag{8.5}$$

$$u_{2,2}\psi_2 + u_{3,2}\psi_3 + u_{4,2}\psi_4 = b_2 \tag{8.6}$$

$$u_{3,3}\psi_3 + u_{4,3}\psi_4 = b_3 \tag{8.7}$$

$$u_{4,4}\psi_4 = b_4 \tag{8.8}$$

It is simple therefore to find $\psi_4 = b_4/u_{4,4}$. This result may then be propagated through the higher rows of the matrix and the right hand side vector (8.5)–(8.7), producing zeros in the right most column of the matrix. The process continues to next find ψ_3 and the remaining unknowns in a similar manner.

As an alternative to using the LU approach, a direct Gaussian elimination routine was written to solve the system by hand. Gaussian elimination subtracts multiples of rows from lower rows in the matrix in turn to form an upper triangular matrix. The same operation is also applied to the right hand side vector b. The system can then be solved using backwards substitution as detailed in Figure 8.1. This approach avoids a full LU factorisation. It is well known that there are issues with the numerical stability of using Gaussian elimination for the solution of linear systems. The issues arise from floating point round off after division by a small number, a phenomenon which may occur as multiples

$$\begin{array}{|c|c|c|c|} \hline u_{1,1} & u_{2,1} & u_{3,1} & u_{4,1} \\ \hline & u_{2,2} & u_{3,2} & u_{4,2} \\ \cline{2-4} & & u_{3,3} & u_{4,3} \\ \cline{3-4} & & & u_{4,4} \\ \cline{4-4} \end{array} \times \begin{array}{|c|} \hline \psi_1 \\ \hline \psi_2 \\ \hline \psi_3 \\ \hline \psi_4 \\ \hline \end{array} = \begin{array}{|c|} \hline b_1 \\ \hline b_2 \\ \hline b_3 \\ \hline b_4 \\ \hline \end{array}$$

Figure 8.1: Illustration of solving a linear system for an upper triangular matrix

of rows of the matrix are subtracted from subsequent rows in order to generate leading zeros. However for this implementation, no issues have been noticed for the matrices assembled for the different inputs in SNAP, with both this method and the library alternative producing the same numerical answers. Note that this does not guarantee that issues may not occur for all problem inputs.

Vectorisation of the Gaussian elimination has been ensured via the use of an OpenMP `simd` compiler directive over the inner-most loop over matrix elements in each row, so that the multiply-subtract of rows may be executed in parallel as SIMD instructions. The vectorisation of the backwards substitution step is again performed automatically by the compiler, and occurs on the inner-most loop zeroing columns and updating the right hand side vector. For the first of such iterations only one vector lane may be used due to the backwards substitution algorithm updating only the last element of the vector; subsequent iterations increase the number of update positions by one each time. As the matrix is only 8-by-8 this does limit the possible effectiveness of vectorisation depending on the vector width. This is particularly acute on GPUs, where on NVIDIA architectures the vector width is 32 (according to a warp), and launching only enough work for 8 threads per warp will result in poor utilisation of the available resources. Solving multiple matrices per thread block increases the utilisation however causes decreased occupancy and fewer thread blocks running concurrently due to limited shared memory capacity for matrix storage. The issue of assembling and solving these small matrices on GPU architectures requires future work (Section 9.1).

The performance of both methods was tested on a single node consisting of a dual-socket Intel Xeon Gold 6152 (Skylake) CPU, with 22 cores per socket. The following problem size was run, and tested with the `dgesv` routine from MKL 2018 along with an implementation of the Gaussian elimination:

- Mesh size of 32^3 with sides of physical length of 0.1
- 32 energy groups, and 136 angles per octant
- 4 orders of anisotropic moment scattering
- 1 timestep of 0.001s
- Material option 1
- A single inner and outer was run for brevity

Runtime (s)	`dgesv`	Gaussian elimination
Assembly and solve	43.03	14.01
Total	55.49	17.34
Grind time (ns)	47.76	15.20

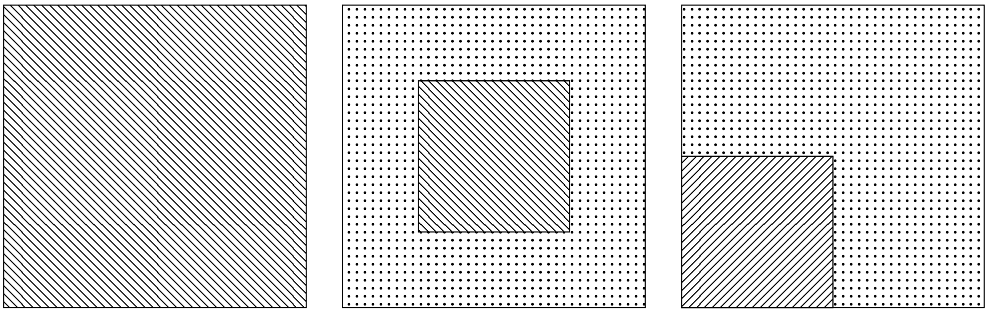
Table 8.1: FEM SNAP runtimes solving systems with MKL and Gaussian elimination

The code was run with 44 MPI ranks (one per core) and no threading enabled inside MKL itself.

The runtimes for the application using both solution methods are shown in Table 8.1. The timings include the assembly and solution time of the matrix, as recording just the solve time alone was found to have a significant overhead. The grind time represents the average (mean) time for solution of one point in the angular flux: the time to calculate the solution in a single cell, for one angle and energy group. The grind time is calculated by dividing the total runtime by the product of the problem dimensions and the number of iterations resulting in an average time for calculation of each problem unknown. The grind time is often used as a figure of merit for the runtime performance of a transport benchmark for procurement (such as APEX, see Section 4.5). It is clear to see that the handwritten Gaussian elimination routine for this 8×8 matrix is much faster than using MKL, on the order of 3X faster despite the potential issues surrounding vectorisation of this method discussed previously. Dense linear algebra libraries are usually optimised for large matrices, certainly larger than 8×8, and the results corroborate that they are not optimised for small matrices. Although batched routines may prove more optimal for small matrices, in the flat MPI regime tested here, the library routine would be used in a non-threaded manner and so the matrices would be solved in turn. Additionally, looking forward to unstructured meshes, it may not be possible to generate a large batch of matrices to solve.

8.3 A practical comparison of the discretisation methods

With an implementation of a FEM port of SNAP, these two methods for discretisation can be compared practically. The comparison will first determine whether the FEM allows the use of a coarser mesh for an equivalent FD solution error; we attempt to identify the corresponding mesh resolutions which lead to a similar quality of solution for both methods. This may be more valid in a real application rather than the proxy application with fictional data, but it is a well known property of higher order methods that coarser cells may be used. The reduction in the number of cells used by the FEM will then determine whether or not the memory footprint of the application may be reduced compared to the FD method. Finally, the FEM requires more steps to produce the solution as it inverts a matrix rather than evaluating simple diamond difference relations, and therefore a brief comparison of the runtime of the applications will be shown.



(a) Layout 0

(b) Layout 1

(c) Layout 2

Figure 8.2: Illustration of SNAP material options

8.3.1 Mesh convergence

In order to determine the number of cells required by the FEM to represent a solution with similar error to FD, the relative error of different mesh resolutions must be compared. As the SNAP benchmark works on fictional data, there is no known solution to compare to. As an alternative, the original SNAP application was run with a fine mesh resolution and the scalar flux and population outputs were used as the baseline solutions against which we can calculate the error.

The SNAP benchmark contains two fictional cross sections and they are auto-generated according to a simple formula based on the input parameters. The SNAP benchmark comes with three different material inputs, with their layouts in 2D shown in Figure 8.2. The first material is shown with lines and has a fixed isotropic source of unity. The second material is shown with dots and has no associated source. Both materials have up, down and within group scattering contributions defined. The first layout uses the first material across the entire domain. The second layout places a cube of the first material in the centre of the domain, with the rest of the domain as the second material. The third layout is similar to the second layout, except the cube of the first material is placed in the corner instead of the centre. All the boundaries are vacuum.

A sample problem was run using both the original FD SNAP as well as the FEM port:

- Cubic domain with sides of physical length 10
- 36 angles per octant (S_{16})
- 16 energy groups
- 4 orders of anisotropic moment scattering
- Convergence criteria of 1.0E-12
- All three material layouts were tested in turn

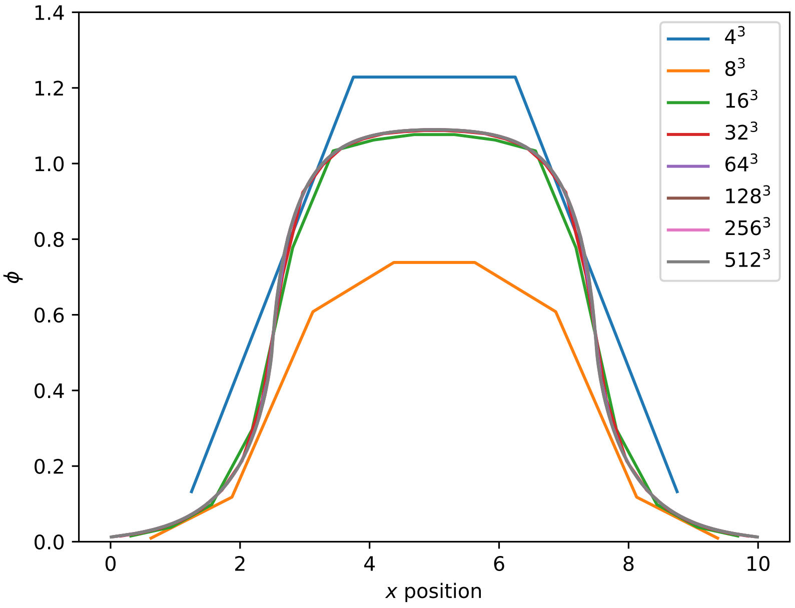
The population count is a measure output by the SNAP benchmark, and is a weighted reduction of the scalar flux over the whole spatial domain, producing a numerical value for each energy group. The population count alone was not sufficient to determine the difference in errors and it was found to hide inaccuracies in the scalar flux solution at both the boundary and at the centre of the problem domain, although this was dependent on the choice of layout and fictional cross sections in the benchmark. In particular, this issue occurs for material layouts 0 and 2, but not for material option 1 (recall Figure 8.2) where it was found that the population count is representative enough of the solution error. Visual inspection of the scalar flux solution for some options available in the SNAP benchmark show that even where the population count changes little with the mesh refinement, noticeable differences in the scalar flux are visible.

To supplement this single error measure, a plane of the scalar flux from the centre of the domain is also used to verify the correct solution. The scalar flux solution for the YZ mid-plane for both the FD and FEM implementations running material layout 1 are shown in Figure 8.3. At a 4^3 mesh, there are 4 cells along this plane; and the vacuum boundary conditions require that the solution is zero on the boundary. The gradient of this zero boundary value in the FD approach forces the scalar flux in the centre of the mesh to be over-estimated. Both methods however tend towards the same scalar flux solution as the mesh is continually refined.

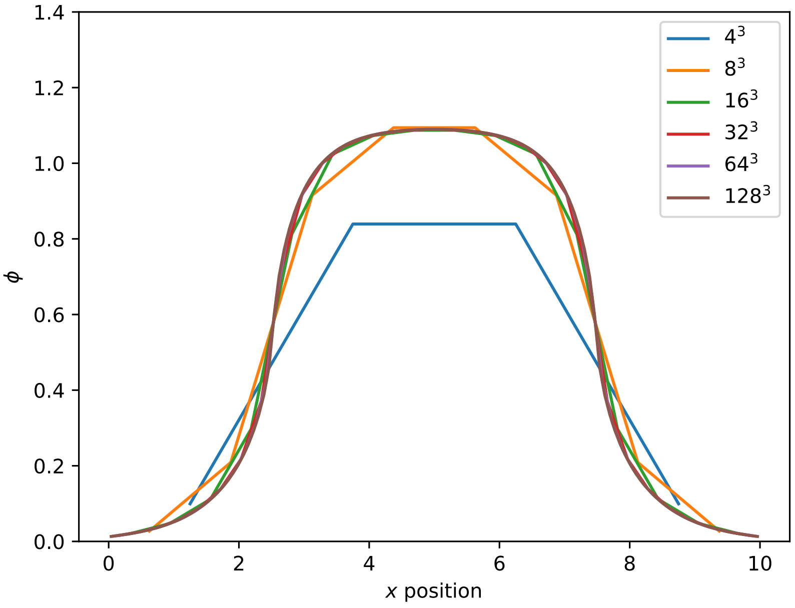
Material layout 0 shows similar behaviour to that of material 1, with both methods converging on the same solution. For material layout 2 however, both methods failed to converge on a solution even at 256^3 cells. This is down to the fictional nature of the data, as common tricks such as also increasing the angular discretisation as suggested by Lewis and Miller did not aid convergence [55]. Therefore material layout 2 shows a limitation of the SNAP mini-app for the purposes of comparing numerical methods, and is excluded from further study.

The baseline 'true' solution is taken as the output from the original (FD) SNAP benchmarking running with a 512^3 mesh. At this resolution the scalar flux solution and therefore the population count changes little with further refinements. The errors in output from the FEM port and coarser FD meshes are compared to this 512^3 solution. The highest energy group is used, along with the YZ mid-plane of the scalar flux, with linear interpolation on the fine solution to compare scalar flux values at the same x-axis points as the coarse solution.

The error for different mesh sizes for material layout 0 are shown in Figure 8.4. The population count (Figure 8.4a) seems to converge for the FD at coarser meshes than the FEM, contrary to the intuition about higher order methods. However plotting the relative mean squared error of the scalar flux mid-plane in Figure 8.4b reveals that this error is lower in the FEM than for the FD method for a given number of cells. Inspection of the scalar flux solution show that the two solutions differ most greatly at the boundaries. Therefore the population count, as a reduction over the entire spatial domain, is smoothing any errors in this boundary region of the scalar flux solution, causing the small numerical values at the boundary (close to zero according to the vacuum conditions) to contribute little to the total population count. It is fair to assume that the scalar flux solution is required in addition to the population count, so for a solution error less than some given value, a FD mesh is required with at least twice as many cells in each axis compared to the FEM mesh. This results in at least an eight times larger grid, meaning that the number of degrees of freedom for both methods for a similar error is equal for this material option in SNAP. This means that both methods have equivalent memory footprints for material



(a) Finite difference



(b) Finite element

Figure 8.3: YZ mid-plane of finite difference and finite element port of SNAP

layout 0.

The error for different mesh sizes for material layout 1 are shown in Figure 8.5. The error in population count (in Figure 8.5a) shows that the FEM is converging faster than the FD method. Therefore this material option in SNAP shows the properties one would expect of higher order methods. For a population solution less than a given error, the gradients of these lines show that the FD approach requires at least four times as many cells in each spatial dimension compared to the FEM approach; this is 64 times more cells in three spatial dimensions. The FEM solution, despite storing eight degrees of freedom per cell compared to one for the FD solution, could result in an eight times memory footprint saving for this problem.

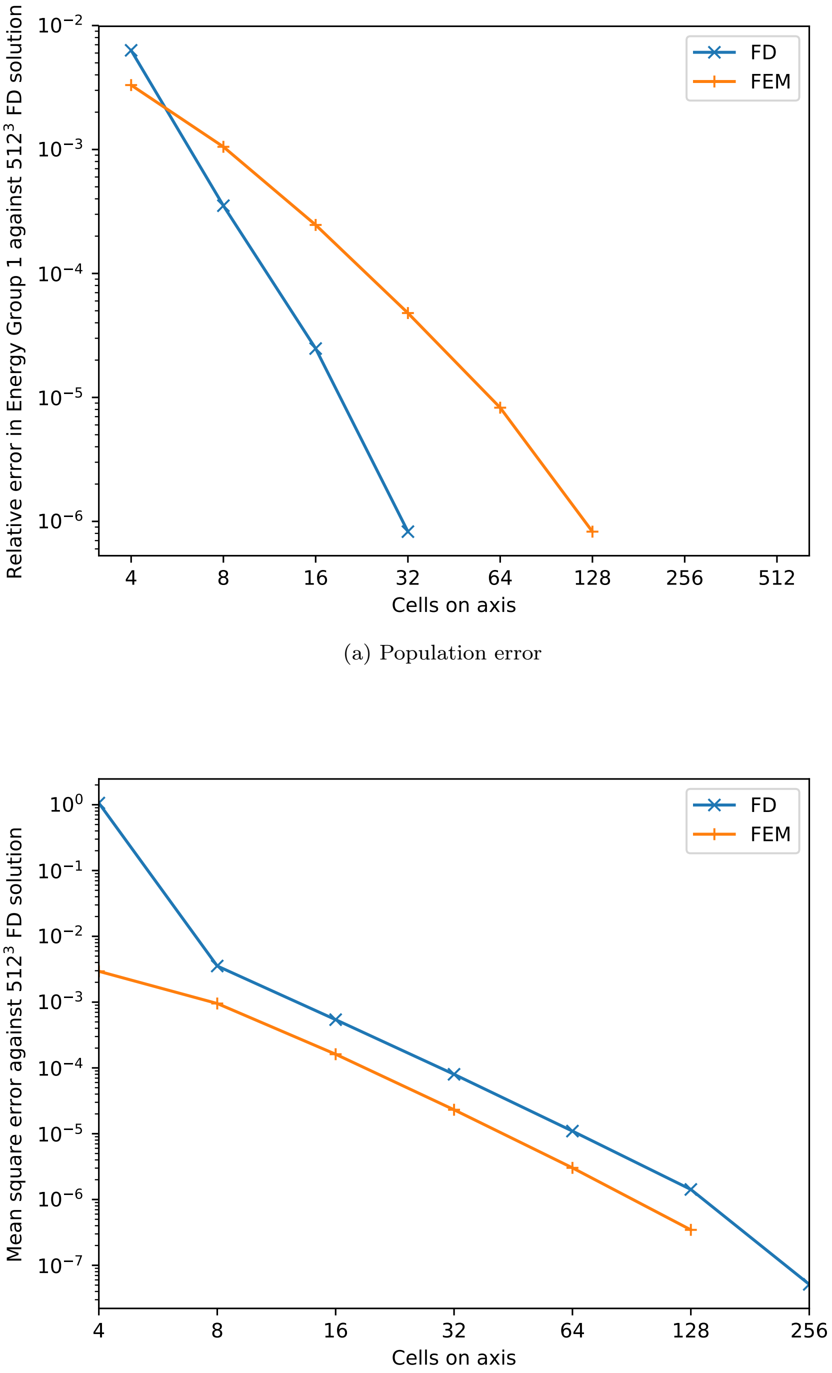
The trade-offs for solution error and mesh resolution for one method over the other depend on what solution is required; the scalar flux solution to the transport equation or the population count, an integrated value based on the scalar flux (see Glossary and Section 8.3.2). The input data will also influence the behaviour, and for the fictional data in the SNAP benchmark this poses a challenge to a more robust investigation. However, the higher order method should require fewer cells, and this was demonstrated for one of the SNAP input options.

8.3.2 Modelling memory capacity

The storage of angular flux dominates the memory footprint of both the FD and FEM implementations. This array is a double precision floating point number for each degree of freedom in the problem; with the linear FEM having eight times the degrees of freedom per cell than the FD discretisation. For material layout 1, it was shown in Section 8.3.1 that the cells may be four times smaller in each dimension for an equivalent solution error. For example a FD mesh of 1024^3 cells may be replaced with a FEM mesh of 256^3 cells. It is therefore simple to model the memory requirement of the angular flux for running a calculation with similar error, which is an allegory of the total memory footprint of the application.

As an example, take a model problem of using S_{32} (136 angles per octant) and 32 energy groups, along with a variety of spatial discretisations. The memory capacity required for different mesh sizes is shown in Figure 8.6, organised in pairs of mesh sizes with similar solution error. Note that the footprint of the FEM mesh is equal to the previous FD mesh shown; for example a 32^3 FEM mesh has the same footprint as a 64^3 FD mesh. The first FD mesh though is 68 GiB in size; and so even though this is a very coarse mesh the footprint is still large; the final FD mesh shown is approximately an exabyte.

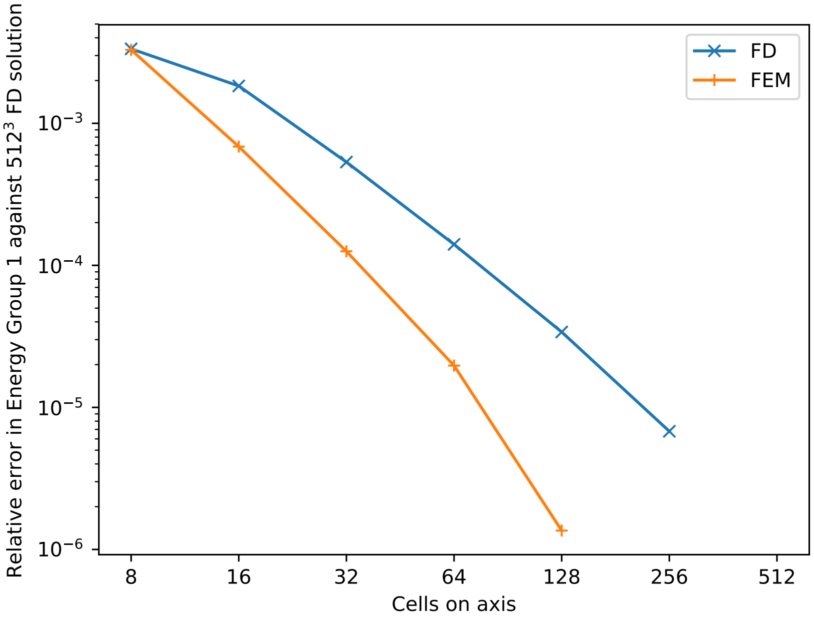
Whilst Figure 8.6 shows the total memory capacity required, and the numbers grow large, it does not highlight the issues surrounding the number of nodes in a supercomputer required just to fit the solution into memory. Two model supercomputers are defined in Table 8.2. The multi-core system is typical of those already found today, with dual-socket nodes with a high number of cores per socket. Typical memory capacity would render 2 GB of standard dynamic random-access memory (DRAM) memory per core. The many-core system on the other hand utilises HBM, which has a limited capacity per socket; and the nodes are single socket. For example, a large portion of the Trinity machine at LANL is made from single-socket Intel Xeon Phi (Knights Landing) (KNL)



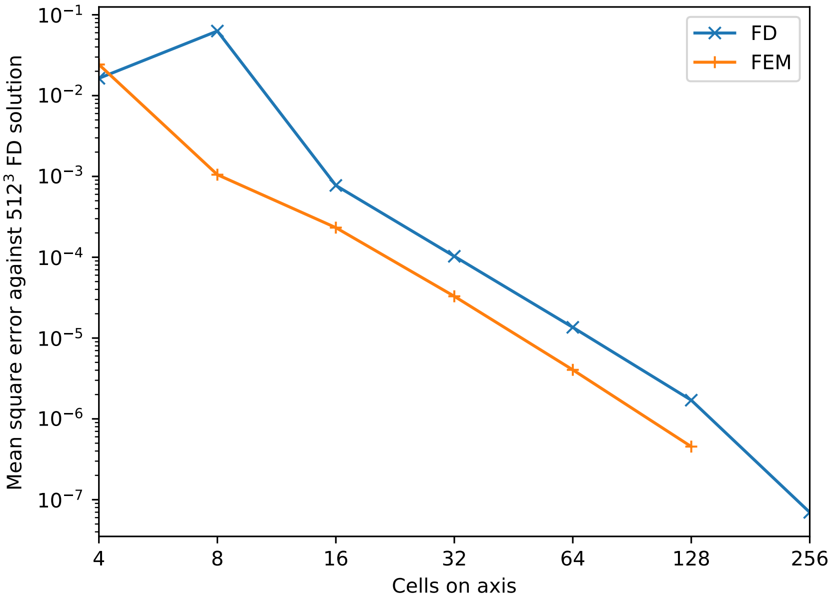
(a) Population error

(b) Scalar flux error

Figure 8.4: Material layout 0 population and scalar flux error



(a) Population error



(b) Scalar flux error

Figure 8.5: Material layout 1 population and scalar flux error

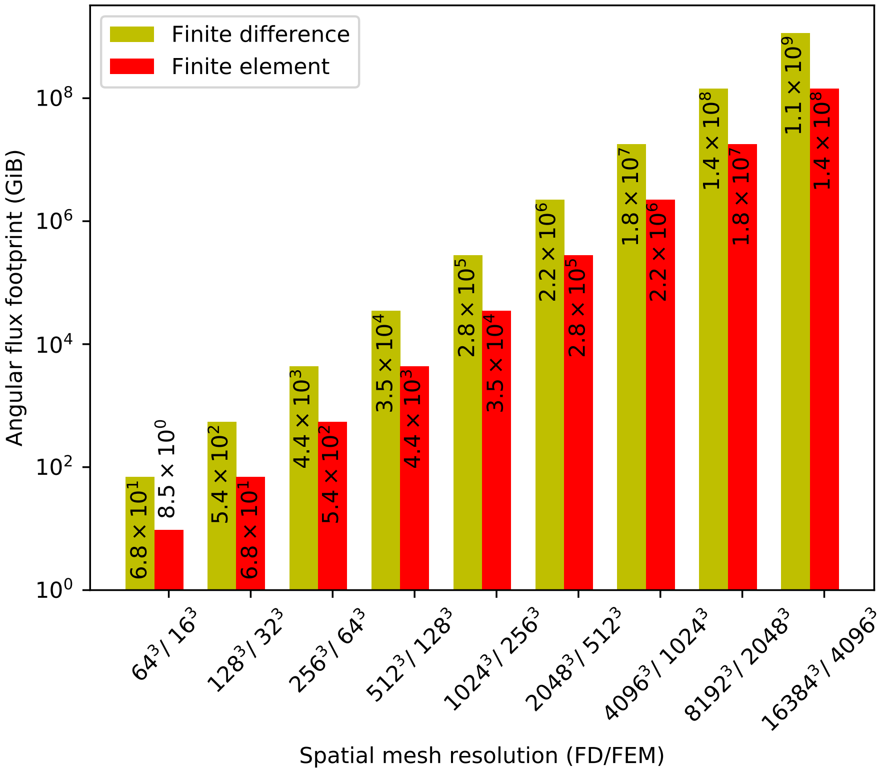


Figure 8.6: Model of memory requirements of the angular flux for finite difference and finite element methods

	Multi-core system	Many-core system
Sockets/node	2	1
Cores/socket	32	>64
Cores/node	64	64+
Memory technology	DRAM	HBM
Memory/node	128 GB	32 GB
Memory/core	2 GB	<0.5 GB

Table 8.2: Hypothetical future multi- and many-core supercomputer nodes

nodes, or else the GPU machine Piz Daint at Swiss National Supercomputing Centre (CSCS) with one GPU per node. Even if multiple GPUs per node are installed, as with the future Sierra (at Lawrence Livermore National Laboratory (LLNL)) and Summit (at Oak Ridge National Laboratory (ORNL)) machines or NVIDIA DGX-1 boxes, the current usage model will be one MPI rank per GPU, and so the memory available to each rank will be determined by the capacity of the HBM on each GPU. The many-core node of Table 8.2 would be logically equivalent from the programmers perspective of MPI ranks and so the model is applicable to such multi-GPU nodes. Note that all current devices utilising HBM offer only 16 GB capacity, and so using 32 GB here represents a future system.

The minimum number of nodes for each system for the model problem used is shown in Figure 8.7. The node requirements for both mesh types are shown overlaid on the same bar. Although this represents just a simple scaling of Figure 8.6, it is the relation between footprint and number of nodes which is of note. For both systems, as using the FEM allows an eight times capacity saving, the minimum number of nodes is eight times less. The number of nodes themselves are however more pertinent. Consider the point on this graph with a 2048^3 FD mesh and 512^3 FEM mesh. This requires 69,632 and 8,704 nodes on the many-core machine respectively. Other than requiring nearly an order of magnitude fewer GPUs to run, the FEM method would allow this computation to fit on near-future machines.

The current fastest GPU machine in the world, Piz Daint at CSCS, consists of 4,256 nodes each containing 1 GPU [91]. The trend for future systems is to decrease the node count and install multiple GPUs per node; the Summit (ORNL) and Sierra (LLNL) machines will contain around 6,400 nodes with multiple GPUs per node. Summit will have 6 GPUs per node [76] with 16 GB of HBM2. Therefore this FEM mesh with a minimum node requirement of 8,704 nodes will indeed fit within the memory capacity of a supercomputer such as Summit where 17,408 GPUs would be required (as the GPUs are 16 GB as compared to the modelled 32 GB capacity); 63% of the machine. The FD mesh would require running on the entirety of Summit in order to fit this problem in GPU memory.

The FEM therefore allows a much more realistic resource to be used for such a computation, possibly even enabling the computation to be performed in the first place. And for more modest mesh sizes, the FEM would allow a user to strong scale the problem using more resource in order to decrease time to solution, something which would not be possible if their maximum allocation is

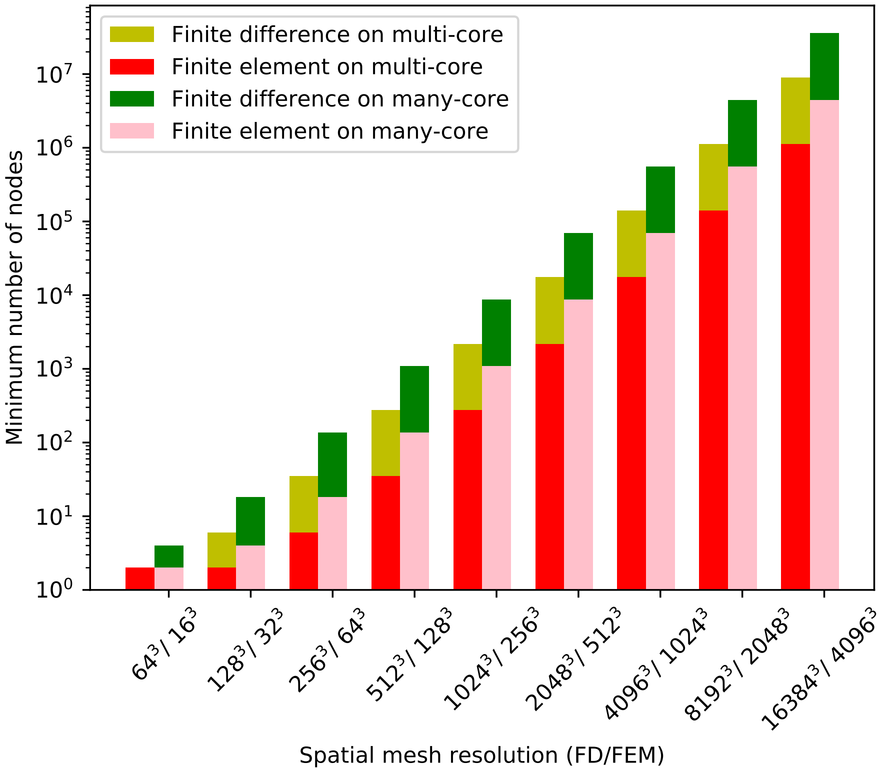


Figure 8.7: Modelled minimum node count for storage of the angular flux

used just to provide sufficient memory capacity.

8.3.3 Runtime implications

In order to assess the time to solution for both methods, an initial study in the runtime of each implementation was conducted. Both codes were run on Swan, a Cray XC40 supercomputer, where each node contains a dual-socket E5-2699 v4 (Broadwell) CPU with 22-cores per socket with 128 GB of memory per node. Both codes were run using flat MPI, with one MPI rank per physical core. Different mesh sizes where chosen, with 36 angles per octant, 16 energy groups and 4 orders of anisotropic moment scattering. The material layout was chosen as in Figure 8.2b. Such modest problem dimensions, particularly with respect to the number of angles and energy groups, was chosen in order to ensure reasonable memory capacity requirements. Both implementations were run until the solutions converged within the specified tolerance. Convergence properties may change as a result of interactions with the angular discretisation and the spatial cell size as discussed in Section 4.3.3, but as with Section 8.3.1 the nature of the fictional data in the mini-app prevents more rigorous treatment of any such issues.

The runtime of a variety of mesh sizes are shown in Table 8.3. A limitation of the original SNAP (FD) application is that it is not able to run on an arbitrary

Mesh size	FD		FEM	
	Cores	Time (s)	Cores	Time (s)
4^3	16	0.1	16	5.0
8^3	32	0.3	22	22.2
16^3	32	1.7	22	139.6
32^3	32	13.7	22	1010.7
64^3	32	102.9	704	229.8
128^3	64	382.6	704	1616.9
256^3	256	789.5	-	-
512^3	1024	1583.6	-	-

Table 8.3: SNAP runtimes for FD and FEM codes running material layout 1

number of cores, and so different cores counts were required; one may also have generated mesh sizes which are multiples of core counts however this is not representative of mesh generation. Core counts were also chosen in order to keep the runtime a reasonable length and are not representative of running on the fewest nodes required to give sufficient memory capacity.

For a fixed mesh size, it is clear that the FEM implementation is much more expensive in terms of runtime than the FD method. The amount of work per grid point is much increased and so this is expected, and for small mesh sizes the runtime disparity is large. However as seen in Section 8.3.1, a coarser mesh resolution may be chosen. For example, a 512^3 FD mesh and a 128^3 FEM mesh both achieved a solution with similar error in a similar runtime of approximately 30 minutes on around 1000 cores. As such, the FEM is competitive with the FD implementation in terms of runtime for a desired solution accuracy. Additionally, this also results in a memory footprint saving due to the coarser mesh as shown in Section 8.3.2.

8.4 Summary

The FEM as a spatial discretisation strategy for the solution of the transport equation was implemented within the SNAP proxy application. This allowed direct comparisons between the original FD and the FEM discretisations in terms of memory footprint and runtime. Although the FEM requires storage for more degrees of freedom per problem unknown, it can still demonstrate memory footprint savings for an equivalent error in solution for some of the artificial problems within the SNAP application. As such it goes some way towards mitigating the memory capacity limitations of advanced architectures which leverage lower capacity, high bandwidth memory technology.

The FEM too allows for an additional source of concurrency in the solution of the transport equation, through the construction and solution of the local matrix in each mesh element. Vector instructions were used to exploit this parallelism in the FEM SNAP implementation. As a precursory study to transport on unstructured meshes, where parallelism in other problem dimensions may be reduced, this may be a vital source of concurrency in order to leverage performance on many-core architectures.

Although there is little motivation from a physical perspective to choose to use FEM over the simpler FD approach on a structured mesh, this chapter shows that such an algorithmic change may be motivated by the architecture of the computational hardware itself. In particular, the FEM gives the application the ability to use a coarser grid which may result in a lower memory footprint and so enable problems to be run in the high bandwidth memory, which have limited capacity in comparison to more traditional memory architectures.

CHAPTER 9

Conclusion

The solution of the deterministic transport equation requires a large amount of computational resource, and so it is therefore imperative that the solver is able to perform well on current and future supercomputers. Disruptive changes to computer architectures are requiring increased levels of parallelism to be found within algorithms and memory bandwidth improvements of the hardware must be exploited for best performance.

This thesis examined the solution of deterministic transport on both multi-core and many-core architectures including GPUs. Computational models were developed in order to both quantify and understand the achieved performance on these devices. This included a memory bandwidth model of the main kernel and a scaling model which also included the communication costs.

State of the art High Performance Computing (HPC) processors are complicated architectures, and whilst a deep understand of their properties is sometimes required to guide optimisations, their function may be simplified when constructing quantitative performance models. In particular the important characteristics for many codes is the relationship between the performance of memory movement and floating point operations. It is precisely this that is captured by the Roofline model [95] (see Section 3.1.1) and also by McCalpin's idea of system balance [68] (see Section 2.1). The Roofline model characterises the hardware, but does not directly assess the performance of an application. For this, quantitative models describing the achieved memory bandwidth of an application code must be derived in the manner shown in this thesis (for example in Section 5.3 and Chapter 6). In most cases, they are treated in a cache oblivious manner, with all reads assumed to come from a single level memory. Whilst in practice this may not occur due to a cache hierarchy for applications with large data footprints (exceeds the size of caches) this is a valid assumption and allows for a much simpler model to be constructed.

Achievable memory bandwidth is typically measured by the gold-standard STREAM benchmark [66], or with codes provided by vendors which do not necessarily measure the same thing in the same way. The STREAM benchmark is really just a proxy application for memory bandwidth bound scientific kernels. This spirit is captured in a portable way in the BabelStream benchmark, which investigates what memory bandwidth is achievable from a range of multi- and many-core devices for the STREAM kernels written in a range of programming models. BabelStream allows quantification of the achievable memory bandwidth from different devices. These results provide scientific application developers with a measure of the maximum attainable memory bandwidth and a baseline comparison with which to compare the performance of the memory bandwidth bound kernels of scientific applications. Such a comparison requires a model of the memory bandwidth of the kernel, and this thesis developed a model for the memory bandwidth of the SNAP sweep kernel. This allowed the efficiency of memory bandwidth utilisation to be quantified.

BabelStream was extended to examine if programming models have any influence on attainable memory bandwidth limits, and the current implementations of all programming models demonstrated a good level of performance portability across a range of devices from a variety of vendors.

The SNAP code from Los Alamos National Laboratory (LANL) is an open-source proxy application used to investigate the solution of the transport equation on modern multi- and many-core architectures [97]. Previous work by Wang et al. on porting this application to GPUs did not result in speedups [94]. In order to leverage the memory bandwidth improvements of GPUs, extra concurrency in the algorithm was sought. By increasing the node-level spatial parallelism within the transport sweep, many-core technologies (and the memory bandwidth advantages that GPUs bring) were demonstrated to be successfully exploited for the first time. The OpenCL implementation of SNAP using an improved concurrent scheme achieved speedups in line with the memory bandwidth improvements GPU devices offer over more traditional multi-core CPU architectures.

This concurrent scheme performed well for a single node, and was again shown to perform well at large scale with scaling experiments conducted on the two largest GPU-enabled supercomputers available: Titan at Oak Ridge National Laboratory (ORNL) and Piz Daint at Swiss National Supercomputing Centre (CSCS).

A performance model to capture the communication and computation time for CPUs and GPUs for the Koch, Baker and Alcouffe (KBA) algorithm was developed. This allowed the scaling performance to be verified, and the runtime improvements were again found to be in line with the memory bandwidth improvements of GPU architectures. When running at scale the KBA schedule does become network bound, however it scales sufficiently well for both multi- and many-core architectures and so continues to provide a viable approach.

Although the high bandwidth memories of GPUs were able to be leveraged for the SNAP proxy application, this was found not to be the case when running on the Intel Xeon Phi (Knights Landing) (KNL) utilising the on package Multi-Channel DRAM (MCDRAM). This is despite having stride one access, effective vectorised code and predictable memory access patterns. The mega-stream mini-app was therefore developed to distil the transport kernel into its most basic form so that memory bandwidth issues were captured on cache-based architectures, along with a performance model of the useful memory bandwidth used by the kernel. The solution of a single chunk under the KBA algorithm highlighted that the memory hierarchy was not being usefully used, and therefore optimisations focusing on ensuring data reuse by controlling which data was being cached together with prefetch instructions allowed performance improvements for this mini-app. These improvements did not leverage the same reductions in runtime for the SNAP proxy application itself, and therefore mega-stream, as a mini-app, does not capture the total picture. This is by design, as a mini-app (by its very nature) excludes detail from the 'parent' application.

To address this the mega-sweep mini-app was designed to capture additional details of the SNAP proxy application. This benchmark took the mega-stream kernel and added the communication and the denominator term. The performance of mega-stream was not optimised to the same degree with the mega-stream optimisations such as non-temporal stores, despite the loop body being almost identical to mega-stream. Preliminary investigations imply that the performance of this application is determined by the cache behaviour rather than main memory bandwidth; this in contrary to the expected limiting factor due to the streaming nature of updating the large angular flux array.

The construction of mini-apps in order to capture the essential behaviour of a larger application has been an important tool to understand the performance profile of the transport algorithm. An important aspect of this is the ability to include a performance model so that a measure of the useful work may be captured. Within the context of transport, this metric was the memory bandwidth. This methodology can clearly be extended to many other codes, where complexity is methodically added to simple loop structures until the performance begins to suffer.

The memory footprint of a transport application is large, filling main memory so as to become memory capacity bound and so the reduced capacity of the high bandwidth memory technologies leveraged by many-core devices provides a limitation. Whilst staging portions of the solution data is natural for the algorithm, this approach shows issues surrounding staging data on the GPUs whilst maintaining good computational performance and low latency communications. The discontinuous Galerkin (DG) finite element method (FEM), a higher order method, provided an algorithmic alternative to reducing the memory footprint of the transport solution. This method also introduces much needed additional concurrency and so may form a viable long term solution if the trend for increased thread count to enable good performance continues.

The performance advantages of advanced many-core architectures can be successfully utilised to improve the performance of the solution of the deterministic Discrete Ordinates (S_n) transport equation. This algorithm does however highlight the complexities of exploiting these improvements, both in terms of expressing concurrency as well as the behaviour of the memory hierarchy.

9.1 Future work

The focus of this thesis has been on solving the transport equation on many-core hardware. A structured, regular, Cartesian mesh was used throughout, resulting in predictable and regular memory access patterns.

There is potential for further investigation into the cache behaviour of the mega-sweep mini-app. This could focus on using hardware counters to help determine which arrays are important to retain in cache for good performance. Additionally, this mini-app could be extended to investigate different communication patterns, including one-sided Message Passing Interface (MPI) style communication patterns which may lower the overheads of communication. Such an implementation would be greatly simplified with some of the new functionality of remote memory access (RMA) notifications due to appear in future MPI versions.

The use of an unstructured mesh greatly increases the challenges for efficiently running an application on many-core architectures. Firstly, the directed graph for a sweep across the mesh may well be unique for each angle; in the 3D structured case all angles within an octant share the same schedule. This graph therefore must be (pre-)calculated efficiently and the parallelism extracted both within and between graphs so that sufficient work can be found to run concurrently.

There are transport solvers and associated research that utilise an unstructured spatial mesh, such as the Tycho 2 mini-app of Section 4.6.7. The focus is generally on how to efficiently schedule the sweep, and does not investigate the solve of each cell, nor does it take a holistic approach to how the solution and the schedule interact depending on the architecture.

The work in Chapter 8 is somewhat a precursor to future study as the DG FEM on hexahedral meshes is a method which may be used to solve the transport equation on an unstructured mesh. Parallelism is available in the construction and solution of the small matrix systems, and therefore it is hoped that the restriction in angular concurrency may be mitigated by this. The viability of using batched BLAS routines could be studied first in the structured case in order to determine if there are any performance benefits before reducing the number of matrices available for parallel computation in a fully unstructured regime.

As an additional complication, a general unstructured mesh may cause a cycle within the graph/sweep schedule. These cycles may exist across process boundaries and/or may be localised to a single processor. Therefore an efficient way to break cycles in parallel should be sought, while maintaining stability of the iterations and a small iteration count (in terms of the numerical method), but also ensure spatial parallelism and minimal global communication to break such cycles. The mesh may move for each timestep and so whatever approach is taken to remove cycles must be applied regularly.

It would be important to again consider a range of many-core devices, including CPUs and GPUs, from multiple vendors so that a portable solution may be sought. This could firstly be to see if the memory bandwidth improvements such devices offer might be leveraged for unstructured mesh transport. Additionally devices are becoming available with dedicated hardware resources for small matrix operations, designed to target machine learning applications. It would be of great interest to see if their reduced precision but potentially increased throughput may be a viable compute accelerator for HPC applications of this nature.

Some future network interconnects promise integration with the processor package resulting in reduced latency. The work in Chapter 7 showed that even at modest scale much of the runtime is a result of MPI communications. Investigating how the properties of these new interconnects benefit (or otherwise) transport is also important.

Appendices

APPENDIX A

Applying the finite difference method to the transport equation

In this appendix, a brief introduction to the finite difference (FD) method will be given. The method will then be applied to discretise the spatial dimension of the Discrete Ordinates (S_n) transport equation. The application of the sweep dependency via upwinding will also be shown. Although much of this material can be found in the textbook of Lewis and Miller [55], it is convenient to include within this thesis especially for the comparison with the finite element method (FEM) in Section 8.1.

A.1 Diamond difference relations

As discussed in Section 4.2.1, the continuous spatial domain may be divided into discrete regular regions called cells. In two dimensions these regions will be squares (or rectangles) and in three dimensions they will be cubes (or cuboids). Note that this thesis only considers regular Cartesian grids. The equation of interest (in this thesis this is the S_n transport equation) may then be calculated at discrete locations: the centre of each cell and the centre of each face (or edge in two dimensions).

The central diamond difference equation relates the solution in the centre of the cell with the solution at the centre of the cell faces. In three spatial dimensions, an equation is introduced for each face pair:

$$\psi(x_{i,j,k}) = \frac{\psi(x_{i-1/2,j,k}) + \psi(x_{i+1/2,j,k})}{2} \tag{A.1a}$$

$$\psi(x_{i,j,k}) = \frac{\psi(x_{i,j-1/2,k}) + \psi(x_{i,j+1/2,k})}{2} \tag{A.1b}$$

$$\psi(x_{i,j,k}) = \frac{\psi(x_{i,j,k-1/2}) + \psi(x_{i,j,k+1/2})}{2} \tag{A.1c}$$

When appropriate rearrangements according to the angular directions of the

S_n quadrature set, these equations describe the outgoing face flux in terms of the incoming and cell centred flux solutions. This is also pertinent to the boundary conditions, where only edge values (those marked with a ±1/2) are given.

A.2 A finite difference discretisation of the transport equation

The discretisation of the transport equation using the central finite difference (FD) equations of (A.1) requires rearranging these FD equations depending on the octant that each particular S_n angular direction belongs. For simplicity, the stationary transport equation will be discretised in this section in one spatial dimension, although it is readily extended to higher dimensions and a time dependent representation.

The transport equation in one dimension may be written as follows, where the right hand side of the equation is simplified:

$$\hat{\Omega}\nabla\psi(x) + \sigma(x)\psi(x) = S(x) \tag{A.2}$$

In this case, the angular direction $\hat{\Omega} = \mu$, and so therefore the streaming operator may be written explicitly as a differential:

$$\mu\frac{d}{dx}\psi(x) + \sigma(x)\psi(x) = S(x) \tag{A.3}$$

The continuous angular domain may then be discretised by integrating this over the spatial domain of a single cell $[x_{i-1/2}, x_{i+1/2}]$. Some approximations are made during this step; notably the cross section is assumed to be piece-wise constant across the cells and so

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \sigma(x)\psi(x)dx \approx \sigma(x_i)\Delta_i\psi(x_i) \tag{A.4}$$

where $\Delta_i = x_{i+1/2} - x_{i-1/2}$, the width of the cell. A similar approximation is made to the right hand side S.

Integrating (A.3) over the spatial cell therefore yields:

$$\mu\left(\psi(x_{i+1/2}) - \psi(x_{i-1/2})\right) + \sigma(x_i)\Delta_i\psi(x_i) = \Delta_i S(x_i) \tag{A.5}$$

where the usual evaluation of a definite integral has been followed for the streaming operator and the approximation (A.4) used for the collision term.

In one dimension, (A.1a) might be rearranged in the following two ways (as was shown in (4.4)–(4.5)):

$$\psi(x_{i+1/2}) = 2\psi(x_i) - \psi(x_{i-1/2}) \tag{A.6}$$

$$\psi(x_{i-1/2}) = 2\psi(x_i) - \psi(x_{i+1/2}) \tag{A.7}$$

Let $\mu > 0$, whence the leftmost cell boundary $\psi(x_{i-1/2})$ is known. The following may symmetrically followed in the case of $\mu < 0$. As such, (A.6) may be substituted into the transport equation (A.5) so as to eliminate the outgoing flux on the cell $\psi(x_{i+1/2})$:

$$2\mu\left(\psi(x_i) - \psi(x_{i-1/2})\right) + \sigma(x_i)\Delta_i\psi(x_i) = \Delta_i S(x_i) \tag{A.8}$$

This process created the *upwind* dependency which results in the need for the wavefront sweep across the spatial domain according to the angular direction.

Finally, the cell centred value $\psi(x_i)$ may be found by rearranging this equation, and the outgoing flux calculated using (A.6):

$$\psi(x_i) = \frac{\Delta_i}{2\mu} S(x_i) + \psi(x_{i-1/2}) - \frac{\Delta_i}{2\mu} \sigma(x_i)\psi(x_i) \tag{A.9}$$

This is usually further rearranged so that the $\frac{\Delta_i}{2\mu}$ term may be a constant factor:

$$\psi(x_i) = \frac{\Delta_i}{2\mu} \left(S(x_i) - \sigma(x_i)\psi(x_i) + \frac{2\mu}{\Delta_i}\psi(x_{i-1/2}) \right) \tag{A.10a}$$

$$\psi(x_{i+1/2}) = 2\psi(x_i) - \psi(x_{i-1/2}) \tag{A.10b}$$

This final equation (A.10) shows that an update to the angular flux ψ for a single angle and energy group in a cell may be calculated as a simple addition of the terms, with the outgoing flux for the neighbouring cells subsequently calculated with a simple diamond difference relation. This forms the heart of the computational kernel which sits in the loop structure of the full solver as shown in Section 4.3.

APPENDIX B

Applying the finite element method to the transport equation

In this appendix, a brief introduction to the FEM will be given. The method will then be applied to discretise the spatial dimension of the S_n transport equation. The application of the sweep dependancy via upwinding will also be shown.

B.1 Test functions

The FEM was first described by Reed and Hill in order to solve the transport equation on triangular meshes [85]. They intuitively wanted to express the angular flux solution in each cell as a polynomial instead of a single value like the FD method. The method has been more rigorously defined subsequently.

In order to construct the element, a set of basis functions are chosen, and the solution is represented as a linear combination of these functions; here we focus on a polynomial representation of the solution in the cell. These functions are known as trial or test functions in the formal discretisation method; it is often convenient to use the same functions for both the trial and test bases.

For our particular choice of finite element space, each function is associated with a node, which is a point in the cell. By definition the functions are chosen so that they evaluate to 1 only at the associated node and 0 at all other nodes. Note they may take non-zero values at other points in the element. For a linear basis (the lowest order), the nodes are positioned on each vertex of the cell. The 2D linear element in Figure B.1 has four functions, with each associated to a corner.

As discontinuous Galerkin (DG) finite elements are used to discretise the spatial domain, nodes are not shared between elements. For nodes which appear on element boundaries or vertices, the neighbouring element also contains a node at the same physical location. For example, consider a regular structured mesh of elements such as that shown previously in Figure B.1. Four such elements are shown in Figure B.2. Note that the point at which all four cells meet

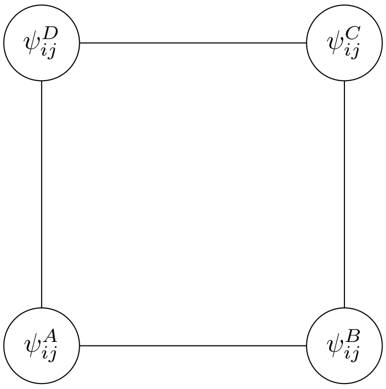


Figure B.1: A 2D linear element with basis functions at associated vertices

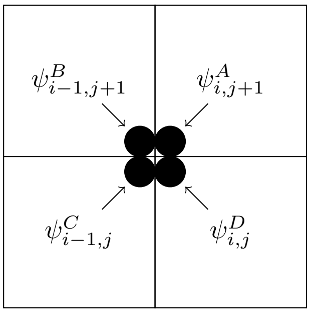


Figure B.2: Illustration of distinct nodes in the discontinuous Galerkin finite element method

occurs at some physical position in space and there are four values representing the flux at this point given by a node positioned on a vertex in each element. With sufficient mesh resolution and convergence of the numerical method, all nodes at the physical location should be equal. However the method allows discontinuities between element boundaries.

We begin by describing the method in one spatial dimension. Simple 1D linear test functions are chosen for a reference element, a line defined on the range $[-1, 1]$:

$$v_{+1}(x) = \frac{1+x}{2} \tag{B.1}$$

$$v_{-1}(x) = \frac{1-x}{2} \tag{B.2}$$

The nodes here are at the end points of the line, $x = -1$ and $x = 1$. Note that v_{+1} is equal to 1 when $x = 1$ and equal to zero at the other node point $x = -1$.

These 1D linear basis functions v are shown in Figure B.3 as dashed lines for the reference element. For the finite element approximation the value in this 1D cell is then represented as a linear combination of these basis functions:

$$\psi(x) \approx \psi_1 u_1(x) + \psi_2 u_2(x) \tag{B.3}$$

This is shown in Figure B.3 as a solid line, where the coefficients have been

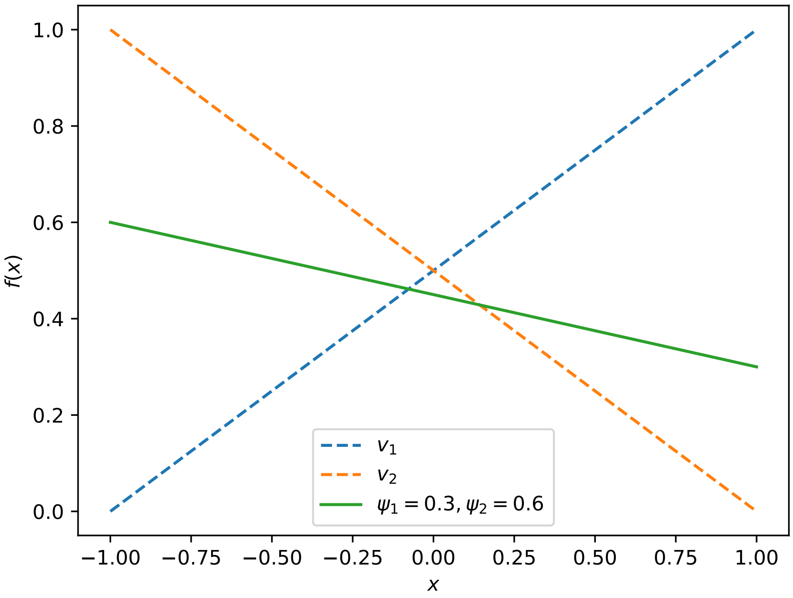


Figure B.3: 1D basis functions

chosen to be 0.3 and 0.6. The FD discretisation gives a single value at the centre of the cell $x = 0$; in this example it would be 0.45. The FEM discretisation instead describes the gradient through the cell.

These 1D functions may be combined in a simple way to generate a basis of test (and trial) functions for 2 and 3 dimensional elements by taking products of the 1D functions. For example, a 3D function associated with the vertex $(-1, 1, -1)$ on the reference element cube may be chosen as:

$$v_{-1,+1,-1}(x, y, z) = v_{-1}(x)v_{+1}(y)v_{-1}(z) \tag{B.4}$$

Higher order basis than linear may also be chosen, which gives rise to more node points in each element. A quadratic basis for the 1D element adds an additional point in the centre of the line. This gives rise to 9 nodes in a 2D element and 27 in a 3D element. This allows for a more complex gradient to be described throughout the cell, and therefore potentially gives a higher order accuracy solution for problems with smooth spatial variations.

B.2 A finite element discretisation of the transport equation

Although much has been written about the FEM, and in particular the DG variant, it is usually discussed in the general sense. We now need to specialise the method to the equation that one wishes to solve and the application to the transport equation additionally involves an upwind dependency (as with the standard FD discretisation) which arises from the treatment of the streaming operator.

Recall that the transport equation (4.1) from Chapter 4 was expressed in a time dependent form. For simplicity the 2D stationary variant (B.5) will be used in this chapter, and the source term simplified. It may be extended to 3D, and indeed the implementation will use the 3D version.

$$\hat{\Omega}\nabla\psi(x,y) + \sigma(x,y)\psi(x,y) = S(x,y) \tag{B.5}$$

The DG FEM method may be applied to this equation as follows. The angular and energy domains are discretised as usual (see Section 4.2.2 and Section 4.2.3). The spatial domain is split into a number of elements Ω. Note that Ω is the common representation for an element and differs from the angle $\hat{\Omega}$. Firstly, multiply by the test function v and integrate over the area of an element:

$$\int_\Omega \hat{\Omega}\nabla\psi(x,y)v + \int_\Omega \sigma(x,y)\psi(x,y)v = \int_\Omega S(x,y)v \tag{B.6}$$

This may be further evaluated by integration by parts, using n for the outward facing normal vectors:

$$-\int_\Omega \hat{\Omega}\nabla v\psi(x,y) + \int_{\partial\Omega} \hat{\Omega}\cdot n\psi(x,y)v + \int_\Omega \sigma(x,y)\psi(x,y)v = \int_\Omega S(x,y)v \tag{B.7}$$

It is convenient at this stage to expand the S_n term $\hat{\Omega}\nabla$:

$$\begin{aligned} -\int_\Omega \mu\frac{\partial v}{\partial x}\psi(x,y) - \int_\Omega \eta\frac{\partial v}{\partial y}\psi(x,y) + \\ \int_{\partial\Omega} \hat{\Omega}\cdot n\psi(x,y)v + \int_\Omega \sigma(x,y)\psi(x,y)v = \int_\Omega S(x,y)v \end{aligned} \tag{B.8}$$

The angular flux ψ is approximated by a linear combination of trial functions u:

$$\psi(x,y) \approx \sum_{i=1}^{K} \tilde{\psi}_i u_i(x,y) \tag{B.9}$$

Each $\tilde{\psi}$ is a single coefficient which corresponds to a node point in the element (cell). For a linear discretisation all node points occur on the vertex (corners) of the element, and in 2D $K = 4$.

In order to evaluate the integral over the boundary term $\int_{\partial\Omega}$, the upwind dependency must be introduced in order to maximise the numerical stability of the method. Only values on incoming face boundaries are known during the sweep, and therefore downwind faces must be treated as unknowns in the equation. The incoming surfaces are denoted as $\partial\Omega^-$ and the outgoing surfaces $\partial\Omega^+$. The specific faces on the element to which these surfaces relate depends on the sweep direction. Incoming surfaces occur when $\hat{\Omega}\cdot n < 0$, and outgoing when $\hat{\Omega}\cdot n > 0$. On incoming surfaces the upwind values correspond to known values from neighbouring elements and so these are placed (as a constant term) on the right hand side of the equation:

$$\begin{aligned} -\int_\Omega \mu\frac{\partial v}{\partial x}\psi(x,y) - \int_\Omega \eta\frac{\partial v}{\partial y}\psi(x,y) + \int_{\partial\Omega^+} \hat{\Omega}\cdot n\psi(x,y)v + \int_\Omega \sigma(x,y)\psi(x,y)v = \\ \int_\Omega S(x,y)v - \int_{\partial\Omega^-} \hat{\Omega}\cdot n\psi(x^-,y^-)v \end{aligned} \tag{B.10}$$

Note that the ψ on the right hand side is taken from the neighbouring element (x^-, y^-).

The equation (B.10) is expressed using integrals, however these cannot in general be known exactly. These integrals are therefore calculated with an approximation, for example a Gaussian quadrature rule is used. Note that this is a different quadrature to the S_n quadrature set chosen to approximate the scalar flux (defined as the integral over the angular dimension of the angular flux). When using the test functions as described in Section B.1 it is possible to evaluate the integrals exactly using sufficient points in the Gaussian quadrature rule. In general the Gaussian quadrature rule allows approximation of an integral using Q points x_q with associated weights w_q:

$$\int_\Omega f(x) = \sum_{q=1}^{Q} w_q f(x_q) \tag{B.11}$$

The Gaussian quadrature of (B.11) and the angular flux approximation (B.9) are applied to (B.10) to give the computable form of the transport equation under the FEM. This substitution results in an equation for each v_j. Although not shown here, note that the source term S contains a contribution from the angular flux as part of the scattering term and therefore the approximation for ψ must also be applied there.

The result is a system of K equations with K unknowns (the $\tilde{\psi}$), with K equal to the number of nodes in the element. The equation for a single element (B.10) can be expressed as a small $K \times K$ linear system $A\tilde{\psi} = b$, after evaluation of all the integrals. The matrix is constructed by looping over the nodes and faces in the element and adding contributions to the appropriate position. A standard Linear Algebra PACKage (LAPACK) `dgesv` may be used to solve this system; the routine typically consists of first performing a lower upper (LU) factorisation of A and then solving the resulting triangular system using forward-backward substitution. Alternatively, it may be inverted directly via Gaussian elimination.

Although the FEM approximation derivation consists of many steps, a computational code deals with the construction and solution of the small matrix in each element, for each angle and energy group. The matrix construction requires access to quadrature and element data in order to evaluate the integrals, and choices should be made as to whether to calculate them on the fly or read them from a precomputed lookup table. The construction must also read neighbouring angular flux data when including the contribution from the boundary integral. Once the matrix is finally constructed, it must be inverted to produce the updated angular flux.

Note that this scheme, as with the FD approximation, sits within the simple iterations on the scattering source, as described previously in Section 4.3. It is not possible to simply replace the FD kernel with a FEM kernel due to the additional angular flux dimension of element nodes. Therefore additional data must be stored per cell, which results in changing a key data structure which must be propagated throughout the code base, including the update of the sources at each node rather than just at each element centre.

Many off-the-shelf FEM solver software libraries then assemble each of these (local) matrices into a large (global) matrix, and use standard linear solvers to invert it. This approach is unsuitable for solving the transport equation because the global matrix would be too large when the full equation is considered. For the streaming-collision operator alone it may not be too onerous as the global matrix may only be 5–7 times the footprint of the angular flux array; however this becomes infeasible to do efficiently with the inclusion of the scattering operator. A (global) matrix-free approach is taken so that only the local matrix is constructed and solved in turn. This also ensures that the outgoing angular flux data for downwind neighbouring elements is also available.

B.2.1 Mapping from the reference element

The preceding FEM discretisation of the transport equation did not include the additional issue of integrating (by Gaussian quadrature) over a general element, and rather assumed that all elements were similar to the reference element.

The basis functions u_i and v_j are defined on the reference element, and must be mapped to a general element. A general element may be formed from the reference element via an isoparametric transform [57]. Such a transform moves the vertices in the element so as to simply move the cell from centring on the origin like the reference element, as well as possibly deforming it, so that it is no longer a regular shape. In 2D, it is convenient to say that the reference element is defined in (x', y') space, and a general element in (x, y) space. In the literature (η, ξ) is often used for this purpose however these symbols are used for the angular cosines in the context of transport and so are avoided here. Recall that the element space was Ω, and so denote the reference element space as Ω'.

Each integral over the general element space can then be rewritten using the Jacobian of the transform of the reference element into a general element:

$$\int_{\Omega} v = \int_{\Omega'} v' \det J \tag{B.12}$$

for general element basis functions v and reference element basis functions v'.

Simple functions are used to map nodes in the reference element $(x', y') \in \Omega'$ to nodes $(x_i, y_i) \in \Omega$ the general element:

$$x(x', y') = \sum_i x_i v_i(x', y') \tag{B.13}$$

$$y(x', y') = \sum_i y_i v_i(x', y') \tag{B.14}$$

Such functions describe the movement of the element verticies into arbitrary positions.

The Jacobian J is therefore the matrix formed of the partial derivatives of the mapping functions:

$$J = \begin{bmatrix} \frac{\partial x}{\partial x'} & \frac{\partial x}{\partial y'} \\ \frac{\partial y}{\partial x'} & \frac{\partial y}{\partial y'} \end{bmatrix} \tag{B.15}$$

The FEM discretised transport equation (B.10) requires partial derivatives of the general element functions over the general element space Ω, e.g. $\frac{\partial v}{\partial x}$. These values are not known directly, and so must be calculated based on the partial derivatives of the reference element. The chain rule is used, utilising terms from the inverse Jacobian; for example:

$$\frac{\partial v}{\partial x} = \frac{\partial v'}{\partial x'} \frac{\partial x'}{\partial x} + \frac{\partial v'}{\partial y'} \frac{\partial y'}{\partial x} \tag{B.16}$$

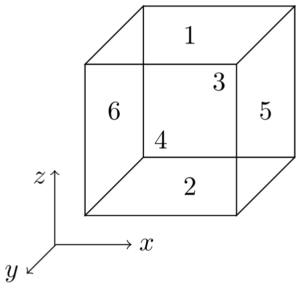


Figure B.4: Faces of a reference hexahedral element centered at the origin

This mapping must be addressed in the matrix construction to ensure that the values are correct for each general cell. In a regular structured mesh, this step may be simplified by dividing by the cell volume. However it is important for future work that the method implemented for this thesis assumes an unstructured mesh and the kernel contains all such operations in the matrix construction.

B.2.2 Calculation of the face normals

The integral over the boundary terms in (B.10) ($\int_{\Omega^+}$ and $\int_{\Omega^-}$) requires the dot product between the S_n angle and the normal vectors of the element. These normal vectors are perpendicular to the face of the element. Whilst this is simple to calculate for a regular shape based on the coordinates of the vertices, in general the faces may be non-flat. As such, the normals are calculated using cross products of the partial derivatives of the coordinate transformation. For a 3D hexahedral element such as that in Figure B.4, the normal for the top face (face 1 in the figure) would be

$$\frac{\partial v}{\partial x'} \times \frac{\partial v}{\partial y'} \tag{B.17}$$

The normals are calculated as the cross product of columns of the Jacobian (B.15). The columns chosen are determined by which dimension is fixed across each face of the reference element, and are listed in Table B.1 for the 3D element in Figure B.4. The ordering of the columns in the cross product operation is important to ensure that the normals are oriented to point away from the element rather than pointing into the element; although it is just the sign of the resulting normal that determines this direction.

The normal vector is of length det J as described above rather than of unit length as expressed in the formal notation, and so therefore mapping from the reference element as in Section B.2.1 for terms which include the normal is unnecessary.

Any precomputation of the element integrals of the basis functions cannot be reused in the calculation of the normal vectors as the integrals for terms requiring the normal are over the element boundary. They can be precomputed separately.

It is a convenient approximation to use the normal from the centre of each face to determine whether the contributions from the face during the matrix assembly contribute as an incoming or outgoing face under the upwinding scheme.

Face	Fixed plane	Cross product of columns
1	$z = +1$	1 and 2
2	$z = -1$	2 and 1
3	$y = -1$	1 and 3
4	$y = +1$	3 and 1
5	$x = +1$	2 and 3
6	$x = -1$	3 and 2

Table B.1: Columns of the Jacobian used in face normal calculation for a 3D hexahedral element

A more rigorous approach would be to evaluate the normal at the quadrature points used to evaluate the integrals over the face, but for curved faces this potentially leads to faces which have both upwind and downwind contributions which has implications for the sweep ordering.

Bibliography

[1] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance, Design, and Autotuning of Batched GEMM for GPUs. volume 9945, pages 21–38. 2016.

[2] M. P. Adams, M. L. Adams, W. D. Hawkins, T. Smith, L. Rauchwerger, N. M. Amato, T. S. Bailey, and R. D. Falgout. Provably optimal parallel transport sweeps on regular grids. *International Conference on Mathematics, Computational Methods & Reactor Physics*, pages 2535–2553, 2013.

[3] M. P. Adams, M. L. Adams, C. N. Mcgraw, A. T. Till, and T. S. Bailey. Provably Optimal Parallel Transport Sweeps with Non-contiguous Partitions. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, pages 1–19, Nashville, Tennessee, 2015. American Nuclear Society.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures - SPAA '95*, 79(44):95–105, 1995.

[5] AMD. OpenCL Optimization Case Study - Simple Reductions. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference — AFIPS '67 (Spring)*, volume 30, page 483, New York, New York, USA, 1967. ACM Press.

[7] D. Appelhans, S. Rennich, A. Kunen, and L. Grinberg. GPU Optimization of the Kripke Neutron-Particle Transport Mini-App. In *GPU Technology Conference*, April 2016.

[8] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. F. Ohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon,

V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical report, NASA, RNR-94-007, 1994.

[9] T. S. Bailey and R. D. Falgout. Analysis of Massively Parallel Discrete-Ordinates Transport Sweep Algorithms with Collisions. In *International Conference on Mathematics, Computational Methods, and Reactor Physics*, pages 1–15, New York, New York, USA, 2009. American Nuclear Society.

[10] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, and W. Joubert. High performance radiation transport simulations: Preparing for TITAN. *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[11] R. Baker and K. Koch. An Sn Algorithm for the Massively Parallel CM-200 Computer. *Nuclear Science and Engineering*, 128(3):312–320, March 1998.

[12] R. Baker, J. McGhee, K. Koch, and J. Morel. Two Sn Algorithms for the Massively Prallel CM-200 Computer. *Submitted to Nuclear Science and Engineering*, 1996.

[13] R. S. Baker. An Sn Algorithm for Modern Architectures. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[14] R. S. Baker, C. R. Ferenbaugh, R. Lally, and J. A. Dahl. Solution of the first-order form of the multi-dimensional discrete ordinates equations on a two level heterogeneous processing system. In *American Nuclear Society Winter Meeting*, volume 105, Washington, DC, 2011.

[15] D. Bednarek, M. Brabec, and M. Krulis. On Parallel Evaluation of Matrix-Based Dynamic Programming Algorithms.

[16] K. Bhat. clpeak. https://github.com/krrishnarraj/clpeak, 2015.

[17] E. Biondo, A. M. Ibrahim, S. W. Mosher, and R. E. Grove. Accelerating Fusion Reactor Neutronics Modeling By Automatic Coupling of Hybrid Monte Carlo / Deterministic Transport on Cad Geometry. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[18] T. G. Bisbas, T. J. Haworth, R. J. R. Williams, J. Mackey, P. Tremblin, A. C. Raga, S. J. Arthur, C. Baczynski, J. E. Dale, T. Frostholm, S. Geen, T. Haugbølle, D. Hubber, I. T. Iliev, R. Kuiper, J. Rosdahl, D. Sullivan, S. Walch, and R. Wünsch. STARBENCH: The D-type expansion of an H II region. *Monthly Notices of the Royal Astronomical Society*, 453(2):1324–1343, 2015.

[19] B. G. Carlson. *Solution of the transport equation by Sn approximations*, page 28. LA (Series) (Los Alamos, N.M.)1891. Los Alamos Scientific Laboratory of the University of California, Los Alamos, N.M., 1955.

[20] H. Courtecuisse and J. Allard. Parallel dense gauss-seidel algorithm on many-core processors. *2009 11th IEEE International Conference on High Performance Computing and Communications, HPCC 2009*, (1):139–147, 2009.

[21] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA, 2010. ACM.

[22] T. Deakin, W. Gaudin, and S. McIntosh-Smith. On the Mitigation of Cache Hostile Memory Access Patterns on Many-Core CPU Architectures. pages 348–362. Springer International Publishing, Frankfurt, 2017.

[23] T. Deakin and S. McIntosh-Smith. GPU-STREAM: Benchmarking the Achievable Memory Bandwidth of Graphics Processing Units (poster). In *Supercomputing*, Austin, Texas, 2015.

[24] T. Deakin, S. McIntosh-Smith, and W. Gaudin. Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport. In *2015 IEEE International Conference on Cluster Computing*, pages 729–737, Chicago, September 2015. IEEE.

[25] T. Deakin, S. McIntosh-Smith, and W. Gaudin. *Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale*, pages 429–448. Springer International Publishing, Cham, 2016.

[26] T. Deakin, S. McIntosh-Smith, M. Martineau, and W. Gaudin. An improved parallelism scheme for deterministic discrete ordinates transport. *International Journal of High Performance Computing Applications*, September 2016.

[27] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. pages 489–507. 2016.

[28] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering*, Special issue (in press), 2017.

[29] T. Deakin, J. Price, and S. McIntosh-Smith. Portable Methods for Measuring Cache Hierarchy Performance (poster). In *Supercomputing*, Denver, Colorado, 2017.

[30] J. J. Dongarra, P. Luszczek, and A. Petite. The LINPACK benchmark: Past, present and future. *Concurrency Computation Practice and Experience*, 15(9):803–820, 2003.

[31] T. Endo and G. Jin. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 132–139, September 2014.

[32] T. M. Evans, A. S. Stafford, R. N. Slaybaugh, and K. T. Clarno. Denovo: A New Three-Dimensional Parallel Discrete Ordinates Code in SCALE. *Nuclear Technology*, 171:171–200, 2010.

[33] J. Freed, S. Gupta, and D. Tiwari. An Analysis of Network Congestion in the Titan Supercomputer's Interconnect (poster). In *Supercomuting*, pages 1–2, 2015.

[34] K. Garrett. A First Look at Performance on the Xeon Phi KNL — Timings from a new mini-app: Tycho 2. In *Department of Energy Center of Excellence Performance Portability Meeting*, April 2016.

[35] J. D. Gelas and I. Cutress. Sizing Up Servers: Intel's Skylake-SP Xeon versus AMD's EPYC 7000 — The Server CPU Battle of the Decade?, 2017.

[36] A. Green, H. Owen, A. Dotti, M. Asai, and A. Aitkenhead. Monte Carlo Validation of Proton Treatment Plans Using Geant4 on Xeon Phi. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, pages 1–9, Nashville, Tennessee, 2015. American Nuclear Society.

[37] A. Greenbaum. *Iterative Methods for Solving Linear Systems.* Society for Industrial and Applied Mathematics, 1997.

[38] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He. SimpleMOC — A PERFORMANCE ABSTRACTION FOR 3D MOC. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[39] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[40] W. D. Hawkins, T. Smith, and M. Adams. Efficient Massively Parallel Transport Sweeps. *Transactions of the American Nuclear Society*, 107, 2012.

[41] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, number November, pages 981–991. IEEE, November 2016.

[42] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[43] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications, 2000.

[44] IBM. *Power ISA Version 2.07*, 2013.

[45] A. M. Ibrahim, D. E. Peplow, and R. E. Grove. Acceleration of Shutdown Dose Rate Monte Carlo Calculations Using the Multi-Step Cadis Hybrid Method. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[46] A. Ilic, F. Pratas, and L. Sousa. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, January 2014.

[47] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[48] J. Jeffers, J. Reinders, and A. Sodani. Chapter 25 — Trinity workloads. In *Intel Xeon Phi Processor High Performance Programming (Second Edition)*, pages 549–579. Morgan Kaufmann, Boston, 2016.

[49] J. Jeffers, J. Reinders, and A. Sodani. Chapter 26 — Quantum chromodynamics. In *Intel Xeon Phi Processor High Performance Programming (Second Edition)*, pages 581–598. Morgan Kaufmann, Boston, 2016.

[50] K. Koch, R. Baker, and R. Alcouffe. Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[51] B. Kochunas and T. Downar. Performace Model Development and Analysis for the 3-D Method of Characteristics. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[52] A. Kochurov and D. Golovashkin. GPU implementation of Jacobi Method and Gauss-Seidel Method for Data Arrays that Exceed GPU-dedicated Memory Size. *Journal of Mathematical Modelling and Algorithms in Operations Research*, 2015.

[53] K. Krommydas, M. Owaida, C. D. Antonopoulos, N. Bellas, and W. C. Feng. On the portability of the OpenCL Dwarfs on fixed and reconfigurable parallel platforms. *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, pages 432–433, 2013.

[54] A. J. Kunen, T. S. Bailey, and P. N. Brown. KRIPKE - A Massively Parallel Transport Mini-app. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, pages 1–13, Nashville, Tennessee, 2015. American Nuclear Society.

[55] E. Lewis and W. J. Miller. *Computational methods of neutron transport.* American Nuclear Society, 1993.

[56] H. Liu, Z. Chen, and X. G. Xu. Monte Carlo calculations of secondary neutron doses in adult male patients during carbon ion radiotherapy. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, pages 1–14, Nashville, Tennessee, 2015. American Nuclear Society.

[57] A. Logg, K.-A. Mardal, and G. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[58] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '14*, 1:219–232, 2014.

[59] M. Martineau and S. McIntosh-Smith. Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 498–508. IEEE, September 2017.

[60] A. Maslowski, M. Sun, A. Wang, I. Davis, T. Wareing, J. Mcghee, G. Failla, and A. Barnett. Acurosсts: a Scatter Prediction Algorithm for Cone-beam Tomography. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, pages 1–15, Nashville, Tennessee, 2015. American Nuclear Society.

[61] Massachusetts Institute of Technology. OpenMoC. https://mit-crpg.github.io/OpenMOC/index.html, 2014.

[62] Massachusetts Institute of Technology. OpenMoC — 2. Method of Characteristics. https://mit-crpg.github.io/OpenMOC/methods/method_of_characteristics.html#, 2014.

[63] Massachusetts Institute of Technology. OpenMoC — 3.3. Transport Sweep Algorithm. https://mit-crpg.github.io/OpenMOC/methods/eigenvalue_calculations.html#transport-sweep-algorithm, 2014.

[64] M. M. Mathis, N. M. Amato, and M. L. Adams. A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations. In *14th ACM International Conference on Supercomputing*, Santa Fe, New Mexico, 2000.

[65] L. Mattes and S. Kofuji. Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. *2010 International Conference on Microwave and Millimeter Wave Technology, ICMMT 2010*, pages 1536–1539, 2010.

[66] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[67] J. D. McCalpin. STREAM2. https://www.cs.virginia.edu/stream/stream2/, 1999.

[68] J. D. McCalpin. Memory Bandwidth and System Balance in HPC Systems. SC'16 Invited Talk, 2016.

[69] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. *Supercomputing*, 8488:53–75, 2014.

[70] L. McVoy and C. Staelin. LMbench3. http://www.bitmover.com/lmbench/.

[71] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[72] D. Miles. When will OpenACC and OpenMP merge. Supercomputing, 2016.

[73] G. Moore. Progress In Digital Integrated Electronics. In *International Electron Devices Meeting*, pages 11–13, September 1975.

[74] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114, September 1965.

[75] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–14. IEEE, April 2008.

[76] Oak Ridge Leadership Computing Facility. Summit. https://www.olcf.ornl.gov/summit/, 2017.

[77] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, M. M. a. Patwary, V. Pirogov, P. Dubey, X. Liu, C. Rosales, C. Mazauric, and C. Daley. Optimizations in a high-performance conjugate gradient benchmark for IA-based multi- and many-core processors. *International Journal of High Performance Computing Applications*, 2015.

[78] S. D. Pautz. An algorithm for parallel Sn Sweeps on Unstructured Meshes. *Nuclear Science and Engineering. and Eng*, 140:111–136, 2002.

[79] S. J. Pennycook, S. D. Hammond, and G. R. Mudalige. Experiences with porting and modelling wavefront algorithms on many-core architectures. 2010.

[80] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the acceleration of wavefront applications using distributed many-core architectures. *Computer Journal*, 55(2):138–153, July 2012.

[81] S. J. Pennycook, J. D. Sewall, and V. W. Lee. A Metric for Performance Portability. In *Programming Models, Benchmarking and Simulation (PMBS) workshop at SC*, pages 1–7, 2016.

[82] S. J. Pennycook, J. D. Sewall, and V. W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, pages 1–12, 2017.

[83] K. Raman, T. Deakin, J. Price, and S. McIntosh-Smith. Improving Achieved Memory Bandwidth from C++ Codes on Intel Xeon Phi Processor (Knights Landing). Presentation at International Xeon Phi User Group Spring Meeting, April 2017.

[84] J. Rawlings and J. Yates. Modelling line profiles in infalling cores. *Monthly Notices of the Royal Astronomical Society*, 326(4):1423–1430, 2001.

[85] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. *Triangular mesh methods for the neutron transport equation*, Technical(LA-UR-73-479):1–23, 1973.

[86] I. Z. Reguly, A.-K. Keita, and M. B. Giles. Benchmarking the IBM Power8 processor. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 61–69, Riverton, NJ, USA, 2015. IBM Corp.

[87] K. Rupp. GPU Memory Bandwidth vs. Thread Blocks (CUDA) / Workgroups (OpenCL), 2016.

[88] Standard Performance Evaluation Corporation. SPEC Accel. https://www.spec.org/accel/, 2016.

[89] E. Strohmaier, H. Simon, J. Dongarra, and M. Meuer. Top 500 - June 2016. http://www.top500.org, 2016.

[90] E. Strohmaier, H. Simon, J. Dongarra, and M. Meuer. Top 500 - November 2016. http://www.top500.org, 2016.

[91] E. Strohmaier, H. Simon, J. Dongarra, and M. Meuer. Top 500 - June 2017. http://www.top500.org, 2017.

[92] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using LogGP. *ACM SIGPLAN Notices*, 34:141–150, 1999.

[93] O. Villa, M. Fatica, N. Gawande, and A. Tumeo. Power/performance tradeoffs of small batched LU based solvers on GPUs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8097 LNCS:813–825, 2013.

[94] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. Scaling the power wall: a path to exascale. In *Supercomputing*, 2014.

[95] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, 2009.

[96] S. Xiao and W. C. Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010.

[97] R. J. Zerr and R. S. Baker. SNAP: SN (Discrete Ordinates) Application Proxy - Proxy Description. Technical report, LA-UR-13-21070, Los Alamos National Labratory, 2013.

[98] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé. Main Memory in HPC. *ACM Transactions on Architecture and Code Optimization*, 14(1):1–26, March 2017.