GPU-Based Acceleration of Miniapp Performance with CUDA

Andrew Lamzed-Short
ID: 1897268

Abstract—**Mini-applications ("miniapps") are small-scale, representative versions of large-scale pieces of scientific- or engineering-focused software that seek to model the performance of an algorithm or program without the associated overhead of executing the larger program. This project seeks to examine the effects of altering the code base of one such miniapp called "SNAP" from utilising only traditional processing cores to incorporating a mixture of traditional computation and offloading applicable workloads to graphical processing cores with the aim to leverage their increased throughput and capabilities for single instruction, multiple data (SIMD) assignments. Ultimately the hope is to examine the performance of the resulting program to ascertain whether the modification is beneficial to the runtime of the algorithm, justifying if this change can be reflected in the larger-scale software SNAP represents.**

To this end, this dissertation details a brief description to the field of High-Performance Computing ("HPC"), along with an introduction and description of the new field of miniapps. Overall aims of the project are outlined before related work in the field of miniapps, application modelling, and other HPC disciplines and projects are presented. This leads into an investigation into the existing SNAP code base and architecture, a thought out and concise approach to and overall design to a potential solution to the problem in hand, with the process finalised by a documented explanation as to the actual implementation. To conclude the report, a testing strategy and results are discussed and analysed, with concluding remarks as to the efficacy and efficiency of the project ending the report.

I. Introduction

A. Background

Modern, frontier-level science calls for large-scale, ambitious projects to answer some of the toughest questions. These projects often involve vast, complex simulations of natural phenomena, from modelling a human brain in one-to-one detail to answer questions about how memory works and how consciousness arises, to modelling the oceans to understand and make predictions about weather and climate change.

One of the predominant questions when designing these simulations is what architecture is best to run this program/suite of programs on. Different workloads and algorithms are designed for and benefit from certain types of computer architecture – some algorithms lend themselves well to being distributed over many cores, whereas others do not. Supercomputers of significant power are leveraged today for the foremost problems of our time: weather simulation and prediction [1], human brain simulation [2], and simulated nuclear weapons testing [3]. The current state-of-the-art supercomputers, their power consumption and performance, are published in a list known as the "Top500" [4], with the most powerful supercomputer to date being "Summit" housed at Oak Ridge National Laboratory, which can reach a performance of 143,500 Tflops/s[1] utilising 2,397,824 processing cores.

In general, supercomputers are comprised of numerous server racks housing many full computer systems – each one containing several CPUs, several graphics card, memory, and high-speed networking capabilities – all interconnected via a high-speed network to allow for communication and cooperation. The topology of the network connecting the computers can vary but two types tend to prevail: computer clusters, and grid computing. Clusters are composed of numerous components that are connected via a centralised resource management system to act as one individual system, with multiple clusters connected by a high-speed local area network (e.g. all in a single site) for low-latency communication; grid computing utilises clusters that are distributed geographically with the underlying assumption that a user of the system need not worry about where the computing resources they are going to be utilising are located – this provides reliability and access to and provision of additional resources on demand. The advantage of cluster computing for supercomputing over grid-based computing systems is stability and very low latency between nodes, as there isn't a need for a high-speed internet connection between sites (also allowing the system to be air-gapped from the outside world for security purposes).

Since the era of Moore's Law with respect to single-threaded/core workloads is coming to an end [5], processors nowadays tend to have multiple cores, with consumer-grade electronics averaging four cores per chip, as can be seen in Figure 1 which details the architecture of a quad-core Intel Core i7 CPU. In addition to hyper-threading (2 threads per physical core), CPUs can have an effective/"logical" core count of twice that. Programming workloads to take advantage of this hardware-based parallelism can be challenging, and parallelising code over multiple nodes in a supercomputer can be even more so.

[1] A "flop" is an abbreviation for 1 floating-point, numerical operation, and a Tflop is a Teraflop, or 10^{12} floating point operations.

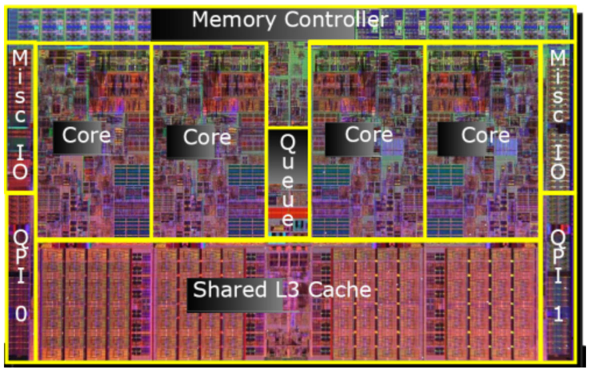

Fig. 1. Quad-core Intel Core i7 CPU Architecture Diagram

This is where libraries such as OpenMP[2] and MPI[3] come in. These are Application Programming Interfaces (APIs) that define how such a complex parallelisation system is to work, and each has multiple open-source implementations that allow for programmers to convert their code from single-threaded to multi-threaded over multiple clusters. It is these technologies predominantly that a large proportion of HPC applications are built with.

Graphical Processing Units (GPU) are a newer technology than CPUs and serve a dedicated purpose of taking instructions from the CPU and performing multiple, hardware-based mathematical operations for translating three-dimensional shapes and coordinates into two-dimensional projections for rendering to a display, and runs multiple small programs called "shaders" to handle colour and lighting. Due to the sheer amount of mathematical calculations that need to be performed to display something onto a display, GPUs have different architecture to that of a CPU. Modern graphics cards, such as Nvidia Turing architecture, pictured in Figure 2, are composed of multiple stream processors, each divided into hundreds of small cores which perform a single integer or floating-point operation. This stream processing approach allows for vast parallel computation over a large dataset in a paradigm called "single instruction multiple data" (SIMD).

This parallelism was previously reserved for image and video processing but a few years ago Nvidia released their CUDA API [6] [7] which allows developers to leverage the stream processing nature of the GPU for general-purpose computation. Scientific workloads from biomedical imaging [8] to deep learning [9] are now done on the GPU, and modern supercomputers, such as Summit, are built with large numbers of GPUs to accelerate workloads and perform previously-impossible simulations and workloads.

Mini-applications ("mini-app") are a new area within the field of High Performance Computing (HPC). These applications are small, self-contained proxies for real applications (typically relating to simulation of physical phenomena) to quickly and concisely explore a parameter space, leading to focused and interesting performance results to investigate potential scaling and run-time issues or trade-offs [10]. Mini-apps capture the behaviour and essence of their parent applications primarily because of two characteristics of many applications running on distributed systems: the performance of an application will mainly be constituted by the performance of a small subset of the code, and many of the physical models that constitute the rest of the application are mathematically distinct and generally have similar performance characteristics [10].

[2]https://www.openmp.org/
[3]https://www.mpi-forum.org/

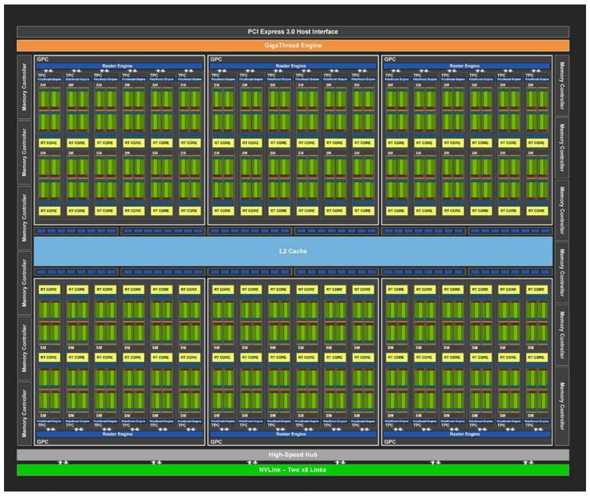

Fig. 2. Nvidia Turing GPU Streaming Multiprocessor Architecture Diagram

B. Objectives

The SN (Discrete Ordinates) Application Proxy (SNAP) is a mini-app that acts as a proxy for discrete ordinates particle transport. It is modelled off another production simulation program developed by the Los Alamos National Laboratory ("LANL") called PARTISN, which solves the linear Boltzmann transport equation (TE)[4], simulating neutron criticality and time-independent neutron leakage problems [11] in a multi-dimensional phase space. SNAP is a proxy to PARTISN because it provides a concise solution to a discretised, approximated version (though with no real-world relevance) of the same problem PARTISN solves, providing the same data layout, the same number of operations, and loads elements into arrays in approximately the same order.

The SNAP algorithm works by defining the phase space as seven dimensions: three in space (x, y, z), two in angle (octants, angles), one in energy (groups, or energy-based bins of particles), and one of time (time step). SNAP sweeps across the spatial mesh, starting in each of the octants proceeding towards the antipodal octant, performing a time-dependent calculation in each cell using

[4]Boltzmann Equation:
https://en.wikipedia.org/wiki/Boltzmann_equation

information from the previous time-step and surrounding cells. This motion forms a wave-front motion that sweeps across the three-dimensional space from corner to corner, with work being divided along each diagonal for parallel execution

With this mini-app in mind, we define three key objectives that the project shall solve. Taken together, these will provide a holistic overview as to the validity and efficacy of this approach of converting CPU-bound parallelised algorithms to utilise the GPU instead (where appropriate). With the SNAP algorithm and open-source repository (specifically the C-based port of the code) in mind, the three objectives are:

- To instrument, profile, and analyse the current implementation of the code in order to identify areas of the code in which it would be applicable and beneficial to convert to CUDA-based parallelisation.
- Using the identified areas found in problem 1, to fork the current C-based port of the SNAP GitHub repository[5] and convert the candidate components and routines from OpenMP to utilise the CUDA libraries instead.
- Following the reimplementation of the algorithm to CUDA technology, the last step is to analyse and evaluate the efficiency and efficacy of the new solution in comparison to the previous CPU-based approach. Ideally, a theoretical maximum efficiency of the approach will also be calculated mathematically, and the actual implementation compared against this as another measure of success.

C. Related Work

A seminal work in the field of mini-apps was written by Heroux et al [10], defining the paradigm. Their Mantevo mini-app suite has show successful development of mini-apps, such as MiniFE for finite element analysis and MiniMD for molecular dynamics simulations, to demonstrate their versatility and applicability. Others have demonstrated such success in other areas, such as Mallinson et al with "CloverLeaf" [12], and Los Alamos National Lab (https://www.lanl.gov/projects/codesign/proxy-apps/lanl/index.php). Mini-apps have been shown to produce similar performance characteristics to their fully-fledged counterparts [10], adding to the efficacy of the paradigm.

General-purpose simulations on GPUs have been studied for a long time, with GPUs being a core part of modern computing clusters [13]. Strong-scaling across multiple GPUs [14] is the ideal approach. Consideration is taken also for conversion of existing code bases [15] and new, bespoke solutions designed with GPU architecture utilisation in mind [14]. Bespoke solutions offer superior code architecture and speed, meaning calculation of theoretical maximum performance increase for a pre-existing code base will have to take this into account.

Writing GPU targeted mini-apps in a developing area of work. Baker et al [16] discuss implementation details of converting the KBA sweep algorithm of the Denovo code system to run on Nvidia Titan GPU. Mallinson et al [12] demonstrate too with CloverLeaf the performance advantages GPU-based architecture targeting can have over purely CPU-based versions. It is important to note that these performance increases might not necessarily be completely reflected in SNAP's algorithm due to other considerations, such as the scaling characteristics of the algorithm [17] and communication technologies as highlighted by Glaser et al [14].

Performance of mini-apps with respect to CPU- and GPU-based parallelisation frameworks have been explored previously and show promising results which add credence to the motivation of this project. Notably Martineau et al [18] reached the conclusion that compiling mini-apps to CUDA resulted in greater efficiencies compared to other targets, though care is needed to consider the implementation (especially with respect to data accesses) to avoid the compiler introducing performance penalties.

Development of the solution must still mimic the behaviour of the original application however, so care must be taken to preserve this. Heroux et al [10] and Messer et al [19] outline the fundamental principles that a mini-app must adhere to and the considerations of forming a mini-app from the base application – all of which would help form testing criteria for this project and future projects to help preserve results and intrinsic behaviour.

II. Project Management

Several factors need to be considered when developing a software project. Management of time constraints is discussed first as producing a novel software project as part of an entire Masters course will run into difficulty - mostly with conflicting or overlapping deadlines but also due to the inevitable problems associated with any software development project. Following on from this, it is also key to highlight how the code was managed and versioned so as to preserve the history of changes and the progression of the project, should ideas need to be experimented with or code recovered should anything unforeseen occur.

A. Time Constraints

Time constraints are always a pressing and important factor to consider when starting a large project. The external largest constraints imposed on this project would be the time required to complete several overlapping coursework assignments for various MSc modules, in addition to setting aside time to adequately prepare for and take the end of year exams in June and July. To manage these obstacles, the timeline presented in Figure 3 was developed early on in the research phase of the project to delineate when and how long certain key stages of development

[5]https://github.com/lanl/SNAP

were to last for, with the potential speed-bumps to this project's progression (i.e. MSc examinations) highlighted and accounted for.

During an early presentation and review of the project, it was made clear that no contingency had been built into the timeline of the project that would alleviate pressure in the event of unfortunate circumstances arriving - whether that be unavailability of resources, issues arising during testing, having to adjust the design, or the typical software development issue of features taking longer to develop than initially estimated. Hence, in Figure 3, contingency periods for overall stages of the project are blocked out in the striped areas shown to allow the project to stay on course and on-time.

B. Source Control

Every major modern software development project uses source control at its core. Source control is key to backing up software in an external repository, maintaining multiple versions of the same code base in a linear or parallelised fashion to be able to restore the code to a former state or maintain a legacy version alongside a more modern version (for instance), and for branching code to allow many developers to work on potentially overlapping areas of the code at once.

To this end, a decision was made to version control the code for the project using `git`. Other source control alternatives were considered but `git` has all of the features discussed in addition to being widely supported, an industry standard, and repository hosting services GitHub and GitLab being two of the most popular offerings available. GitHub offers free hosting for public repositories and is already where the original SNAP repository is hosted.

With this in mind, LANL's SNAP repository was forked into a new repository located at https://www.github.com/alshort/snap. From here it will be cloned onto the local development machine and worked upon in a linear fashion, branching code were experimental code is to be created. Tags will be used to denote stable versions when they are created.

In terms of the specific source control methodology used, the project will be managed using an industry standard git branching model called "git-flow" [20]. The model is illustrated in Figure 4.

The methodology specifically involves a master branch maintained with only tagged releases, and a develop branch that all of the development is done on. Features and hotfixes are done on separate branches and merged back into their parent branch, with testing and release-candidate branches spawned off a fixed point on the develop branch. After complete testing and verification that everything works, they will get merged down into the master branch and a new version tagged.

C. Hardware and Software

Intel's port of SNAP relies on very particular tooling and platform decisions so most of the decisions with regards to testing and development tooling were already determined ahead of time. Furthermore, the decision to utilise CUDA as a platform to accelerate the performance of the mini-app also additional hardware requirements in an Nvidia-based GPU and accompanying drivers and development suites. If the modifications to the code base are successful, it is hoped to execute the software under both a local testing environment and a cluster computing environment at the University of Warwick's Department of Computer Science. Details of both of the set-ups follow.

1) Local Development Platform: It is sufficient enough to be able to compile, execute, and test both the stock SNAP application as well as a GPU-accelerated alteration on a typical home desktop computer as of time of writing, given some particular brand-alignments/proprietary technologies are in mind when the machine is purchased or built. As such, hardware specifications of the home desktop computer used as the local development environment are as follows:

- **Intel® Core$^{\text{TM}}$ i7-6700K CPU**: A single quad-core, consumer-grade, overclockable CPU produced by Intel as part of their 6th generation "Skylake" line of processors. It has a clock speed of 4GHz (overclockable) and comes as standard with hyper-threading enabled for an (apparent) core count of 8.
- **Nvidia GeForce GTX 1070 graphics card**: Pascal architecture-based GPU that supports the CUDA Compute functionality. In terms of hardware specifications, it has a 1.5GHz base/core clock speed, 8GB of GDDR5 memory, and 15 streaming multiprocessors for parallelism.
- **16GB DDR4 RAM**.
- **Samsung 860 EVO SSD**: Storage disk for the Ubuntu operating system the project with be developed and tested on - both the original SNAP porters and cluster environment use the platform and tools on it, so the choice was made mainly for feature-parity and consistency (see below).

This set-up allows for the compilation and testing of the current implementation of SNAP as well as a CUDA-accelerated version by both having the correct hardware that the software is and will be built for, while also being flexible enough in order to test the programs under a range of different inputs and constraints (such as having anywhere from 1 to 8 threads to run the program on). Whilst newer generation Intel processors would be even more ideal, such as the Intel® Core$^{\text{TM}}$ i9-7960X 16-core processor, the current hardware more than suffices enough to yield a satisfactory picture of the performance characteristics of the MPI and OpenMP implementation and limitations for this application.

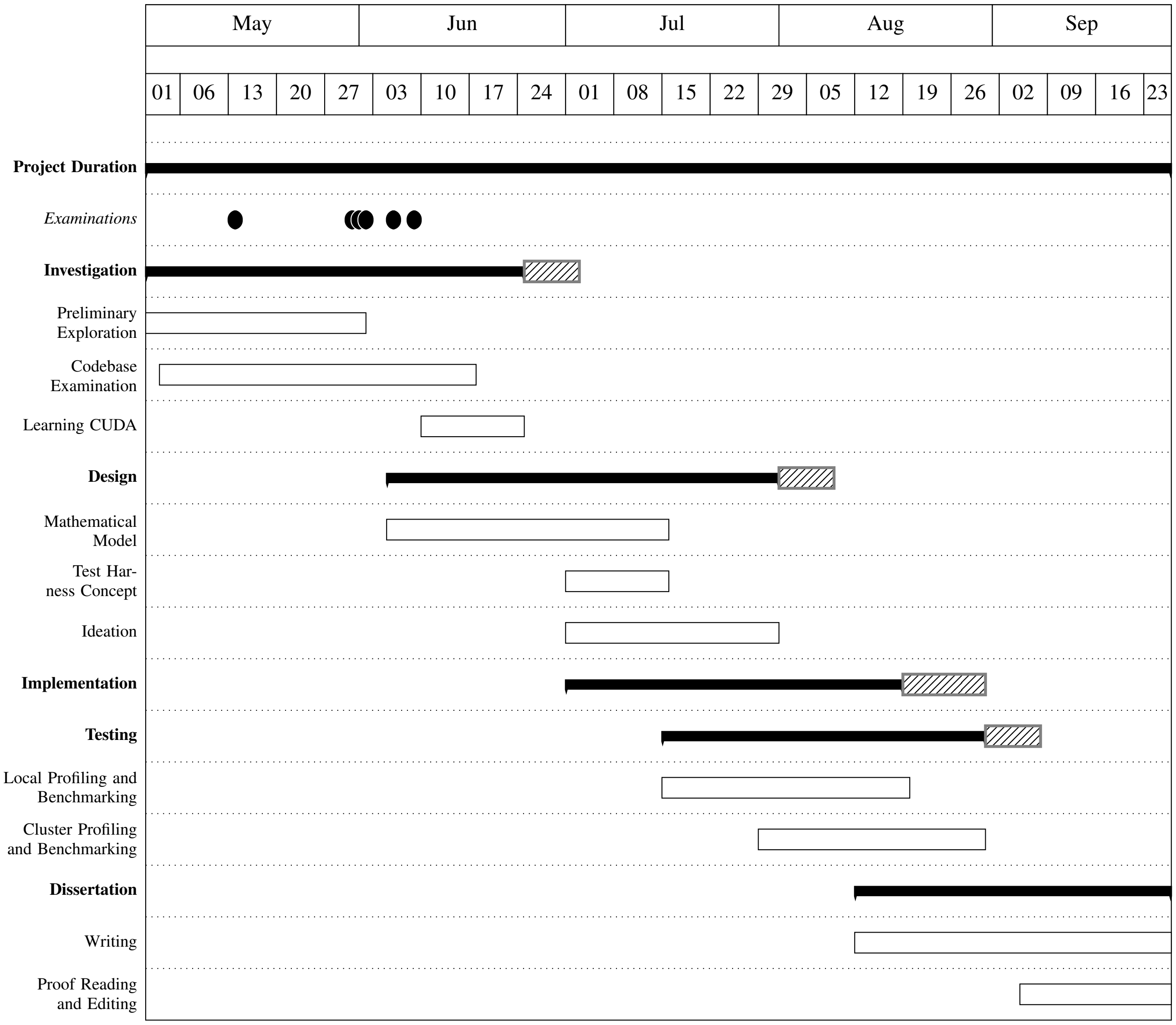

Fig. 3. The Proposed Timeline of this Project

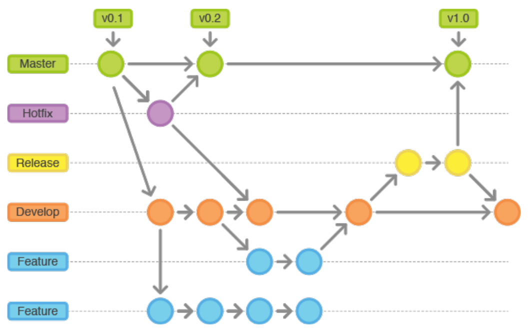

Fig. 4. Visual map of the git flow methodology

2) Computing Clusters: There are two local high-performance computing clusters available for use at the University of Warwick that are ideal to test the potential multi-GPU capabilities of the end solution. Tinis and Orac are both similar in capabilities but differ in terms of architecture and hardware - Orac being comprised of newer hardware whilst Tinis is expected to reach end-of-life in October 2019 [21].

Tinis will be the focus of the cluster-based testing of the GPU-accelerated program because of more than capable hardware to prove or disprove the effectiveness of the final solution, in addition to additional familiarity with it due to the teachings of the CS402 course that is part of the MSc Computer Science program. There are many different types of nodes built into the cluster for handling a wide range of applications: compute nodes, GPU nodes, Phi nodes, and Fat nodes. The Phi and Compute nodes are purely computationally focused, so are not appropriate for the needs of this application (in addition to Phi co-processors being deprecated from Intel's MPI library in the 2018 revision

[22]). GPU nodes will only have the essential hardware in order to operate several graphics devices per node, which again is not appropriate, as this application will need a mixture of CPU and GPU power. Fat nodes have both high-power server processors and Nvidia graphics cards in each node, making them ideal. The specifications for them, as well as the crucial components of the system as a whole, are as follows:

- 4 nodes, each composed of:
  - 32 x Intel Xeon E7-4809 v3 2.0 GHz Haswell cores
  - 1TB DDR3 memory
  - 1 x NVIDIA GRID K2 GPU per node (CUDA capable)
- Networking provided by QLogic TrueScale InfiniBand which supports many-gigabit per second transfer speeds between nodes.
- CentOS 6.x as the underlying operating system. It's a robust, reliable distribution of Linux that incorporates all of the hardware in the cluster under one management system.
- Slurm as the workload manager for submitting the program as a job to any of the available nodes.

In total, this provides, in the best-case scenario, a total of 128 traditional processing cores and 4 high-end graphical processing devices (each with two GK104 GPUs[6]). A hybrid program that has multi-CPU and multi-GPU capabilities should be able to leverage a significant portion of the power provided should the threading and data orchestration work.

Having multiple cores for both types of devices will allow for testing over a range of different configurations of traditional-only computation, and a varying amount of hybrid resources to find the optimal amount as well as any bottlenecks or trends in performance that might appear.

3) Software: Working with an existing and established code base means adhering to existing tool chains where possible in order to correctly compile the software and to have an identical executing environment to run the program in to ensure the program behaves as intended. Thus, most software choices were predetermined for the project. Where nothing was chosen already, the rest of the choices were made due to personal preference, familiarity, or significant advantages were present that made them ideal for their role. All the options are listed below:

- **Ubuntu**: An open-source flavour of Linux favoured by developers for its rich tooling, reliance on shells and terminals and the power they provide, and ability to work at multiple levels (high or low depending upon the circumstances).
- **bash**: A "shell" that provides a command-line interface to execute programs and interact with a Linux-based operating system. Useful for simplicity and for automation purposes with shell scripts, calling make files to handle compilation etc.
- **Visual Studio Code**: A cross-platform, automatable text editor with multiple extensions available. Used specifically in this project for its syntax highlighting capabilities for C, C++, Make, Bash, and LaTeX, as well as executable tasks to automate routines with its built-in terminal, and the latex-workshop extension for dynamically building, parsing, and previewing LaTeXdocuments. The tasks developed for this project can be found in the `.vscode/tasks.json` file located in the source code.
- **mpiicc**: Proprietary compiler for compiling code that uses both the MPI parallelisation specifications and Intel-specific C- and C++-based code.
- **make**: Shell-based program that follows a given set of recipes within a Makefile for compiling multiple dependent parts of source code. Intel provided one to build their SNAP port with their `mpiicc` compiler, and this will further be modified later to incorporate other files and tools.
- **nvcc**: Proprietary compiler produced by Nvidia to compile C++ code that targets the CUDA Compute capabilities of their compatible graphics cards.
- **pdflatex, bibtex**: Back-end programs for Visual Studio Code and latex-workshop to use to built LaTeXand accompanying bibliography files. Errors and output are parsed and displayed in Visual Studio Code's in-built terminal for an enhanced and compact workflow.

Any additional programs used are merely an intrinsic feature of the operating system used. Features like `ssh` to remote into the Tinis computing cluster, `git` for fine-tuned version control, and other utilities are found on most development systems.

III. Investigation

All code previously and actively developed by a third-party is legacy code. SNAP, with its various ports, is no different. The original foundation is developed in FORTRAN90 as this language is still used in many areas of HPC

A. SNAP Repository Cloning and Documentation Review

Initial plans involved gaining a fundamental understanding of the nature of the algorithms and data structures that the SNAP program utilises in order to effectively emulate and model its larger counterpart program. LANL provide the full FORTRAN90 source code, in addition to several ports into other, different languages, in the GitHub repository located at https://www.github.com/lanl/snap/. For the purposes of this project, it will be Intel's C-based port that shall be modified and examined due to ease of use for compilation and finding the appropriate tooling (compilers, syntax highlighters etc.) as well as increased familiarity with the language over FORTRAN90.

[6]As per the specification of the device: https://www.nvidia.com/content/grid/pdf/GRID_K2_BD-06580-001_v02.pdf

The entire repository was forked on GitHub for the purposes of modification. It can be found at https://www.github.com/alshort/snap. All non-C-based ports were removed from the repository in addition to any miscellaneous files and documentation as these were superfluous. Only the `qasnap/` and the `src/` directories (of Intel's code) were kept as these would be the only necessary sets of code needed to build, execute, and test SNAP and this project's proposed modifications.

In the main repository, various presentations and documentation are provided that discuss the reasoning behind creating SNAP, background behind PARTISN and the code SNAP was seeking to model, as well as an overview of some of the implementation details of the main constituent of the approach LANL and Intel took: the "sweep" algorithm.

B. Execution, Profiling, and Investigation of Code

1) Compilation: Intel chose to target the C-based port of SNAP towards Intel-based CPUs, hence the decision, as outlined in section II, to use a machine to locally test the program having an Intel Core i7-6700K CPU. This decision was made due to the original port writers' knowledge of the benefits that the architecture and proprietary improvements could bring to a strictly MPI-based implementation of SNAP. One such proprietary optimisation is the `-xAVX` compiler flag that was introduced from the "Sandy Bridge"-line of Intel CPUs onwards that introduced the option to include new instructions (and expanded old instructions) that allowed for operations - specifically fused multiply-accumulate (FMA) operations - for the execution of some SIMD workloads on floating-point data [23]. This operation is similar in principle and execution to the SIMD nature of data manipulation on a standard GPU but cannot compare to the speed or scale of that which is obtainable on streaming multi-processors.

GNU make is used to compile all facets of the program, with the main compiler being `mpiicc`. Whilst `gcc` is typically used to compile regular, standard C and C++ code, `icc` is Intel's C++ compiler that was built in order to specifically optimise code to run on Intel-created architectures. This includes optimisations for how memory is specifically accessed, how thread-based parallelisation is handled with regards to the OpenMP specification, data layout improvements, and support for modern iterations of C++ language specifications[7]. `mpiicc` is an HPC-specific version of the `icc` compiler that utilises message-passing library built for use with the `icc` program (in particular its C-based capabilities). This library implements the Message Passing Interface (MPI) specification version 3.1 [24] to allow for bi-directional inter-process communication, regardless of how those processes are mapped to the underlying hardware.

[7]https://software.intel.com/en-us/c-compilers

Whilst provided in an operational state, this provision is predicated on the correct resources and dependencies being installed and linked up on the host system. Installation of Intel's MPI libraries is straight-forward but ensuring that they will operate correctly when called can be an issue.

One of the foremost scripts devised when the project started was a bash script that ran several commands in order to register locations and associations of binary files that are a core part of Intel's MPI libraries with the host shell (standard bash in this instance) so that the environment could call upon these when needed by either the compiler or the runtime code. Figure 5 shows this script of "source" commands. The execution of this script was appended to the environment's `.bashrc` file in order for it to get executed before each bash shell has finished initialisation, allowing for the references to be available all to time to accelerate testing and development.

```
#!/bin/sh
. ~/intel/bin/compilervars.sh -arch intel64
    -platform linux
. ~/intel/mkl/bin/mklvars.sh intel64
. ~/intel/bin/iccvars.sh -arch intel64 -
    platform linux
. ~/intel/compilers_and_libraries/linux/mpi
    /intel64/bin/mpivars.sh
```

Fig. 5. intel.sh

2) Understanding the Input Data: Provided as part of the original SNAP repository is a folder called `qasnap`. In this folder resides several input and respective output files for quality assurance and testing. Executing the binary file as above and then comparing it line-by-line to the model output will highlight any discrepancies in the accuracy of the answer - any incorrect modification to the algorithm will flag as an inaccuracy. (The output files also output the timings of the execution but this will be guaranteed to be different on any machine.)

The input data is in a proprietary format in a whitespace-indented file. Due to the fact that the code is a proprietary piece of software developed for a specific task, it is understandable that the documentation is sparse in places, especially with regards to the meaning of these key-value pairs in context.

LANL is in the United States of America, so direct support requests would be costly in terms of time and constrained by time zones, in addition to being potentially low-priority items. Mercifully, within Intel's C-based port contains useful commenting. Hence, alongside analysis and reading of the code, use and meaning of the input variables can be inferred from context for the most part. Figure 6 shows several of the more essential items.

...

3) Execution: Following successful compilation of the SNAP executable, several executions were performed un-

Variable	Use
`nthreads`	Number of threads per process.
`npey`	Number of parallel processing inputs in the y direction.
`npez`	Number of parallel processing inputs in the z direction.
`ndimen`	Total number of dimensions to the grid.

Fig. 6. Important variables defined in the input data

der various environmental-constraints and input data to ascertain the current limitations and performance of the code base as it stands.

Due to its reliance on MPI for its processing, the compiled program is executed via Intel's `mpirun` program at the command line with an argument specifying the size of the MPI communication channel (or overall number of parallelised processes), the `-np` command-line argument, to divide the work into, as opposed to calling the executable by itself. An example execution is as follows:

```
mpirun -np 4 src/snap_mkl_vml --fi qasnap/
    center_src/in01 --fo test.out
```

Figure 7 demonstrates the output of a successful execution of a given input.

```
andy@pc:~/uni/snap/src$ mpirun -np 4 ./
    snap_cpu --fi ../qasnap/center_src/in01
    --fo test.out

*WARNING: PINIT_OMP: NTHREADS>MAX_THREADS;
    reset to MAX_THREADS
Success! Done in a SNAP!
```

Fig. 7. A successful execution of the compiled snap_mkl_vml binary

It was identified early on into the testing - whilst undocumented, it could be said to be a core feature of the MPI specification itself - that defining an `-np` value that is not a whole multiple of the product of the defined grid geometry from the input data (`npey` × `npez`) then the runtime configuration cannot work out how to divide up the workload into whole, viable chunks to send to all processes. Figure 8 shows the error message provided in this scenario.

Ascertaining the characteristics, properties, and limitations of an existing code base before modification provides vital insight that can guide the rest of development. Using the limitation in terms of number of processes to run a sample execution on will guide the test plan in particular (Subsection VI-A).

```
andy@pc:~/uni/snap/src$ mpirun -np 1 ./
    snap_cpu --fi ../qasnap/center_src/in01
    --fo test.out
Abort(202454796) on node 0 (rank 0 in comm
    0): Fatal error in PMPI_Cart_create:
    Invalid argument, error stack:
PMPI_Cart_create(325).....: MPI_Cart_create
    (comm=0x84000002, ndims=2, dims=0
    x7ffd07f649b0, periods=0x7ffd07f649b8,
    reorder=1, comm_cart=0x7ffd07f64bac)
    failed
MPIR_Cart_create_impl(194):
MPIR_Cart_create(58)......: Size of the
    communicator (1) is smaller than the
    size of the Cartesian topology (4)
```

Fig. 8. An unsuccessful execution of the snap_mkl_vml binary due to imposed multi-processing limitations

C. *gprof*

The central issue of profiling parallelised code is that of ascertaining which functions calls are being called and calculating the total amount of time spent there (amongst obtaining other metrics and measurements of interest). In serial code, this task is easy. However, parallel code can be distributed over multiple nodes, processors, and/or threads depending on the resources available, the way the program works, and how the scheduling system decides to allocate work and timeslots. All of this must be orchestrated together, tracked, and aggregated to form one cohesive picture of what the application did on the execution of interest.

`gprof` is a command-line based profiling tool that displays the call graph profile data of any C, Pascal, or Fortran code it was set to monitor[8]. When the compiled binary is run, it will produce a single `gmon.out` file containing the pertinent profile data. This data is consumed by the main gprof application for analysis. In terms of the output it can produce, `gprof` is able to relate the `gmon.out` profile data to the program's symbol table to output a call graph of functions and the functions that call them and how they related to one another. In addition to this, `gprof` can also output the total number of calls of a function and how long the program spend inside each subroutine.

In order to cater for parallel programs, `gprof` is able to sum the results of multiple profile data files into one for a general overview. Producing the multiple traces is performed by setting the `GMON_OUT_PREFIX` environment variable to differentiate the traces. Setting this variable to "`gmon.out-`" will cause `gprof` to append the process's ID to the end of the prefix, producing several files pertaining to the run of the program - one for each process. `gprof -s` will sum these together and produce a useful, informative result.

Performing this profiling on the standard SNAP application compiled earlier in the investigation yield the table

[8]On the provision that the code in question was compiled with the `-pg` parameter beforehand.

shown in Figure 9. In the table is a truncated subset of the five most called and time-intensive subroutines. Multiple executions of the program exhibit near identical characteristics in terms of the subroutines reported back by the `gprof` tool.

% Time	Cumulative Secs	Calls	Name
66.67	0.02	14976	`dim3_sweep`
33.33	0.02	14976	`sweep`
0.00	0.03	29952	`precv_d_3d`
0.00	0.03	29952	`psend_d_3d`
0.00	0.03	14976	`octsweep`

Fig. 9. Amalgamated `gprof` log (truncated to top 5 rows)

Two of the functions are associated with intrinsic data transmission across MPI processes. The others are part of the main transport sweep algorithm that is a central tenant of the SNAP program. This was to be expected but is now experimentally confirmed.

Now that a promising candidate for GPU-acceleration has been found in the `dim3_sweep` subroutine, a review, analysis, and breakdown of the particular code can be done to identify areas for conversion.

1) Code Analysis: `dim3_sweep`

D. Learning CUDA

Being the only way to execute general-purpose computational code on a[n Nvidia] GPU, it is important to comprehensively understand CUDA before delving into any source code modification.

Alongside the content taught in the MSc CS402 High-Performance Computing module, another resource that was used to learn about CUDA was the book "CUDA by Example" (ISBN-13: 978-0-13-138768-3) by Jason Sanders and Edward Kandrot [25], two senior software engineers working on CUDA teams at Nvidia.

As previously mentioned in Section I, GPUs are architected in such a way as to maximise the level of parallelism for a given operation over a given set of data at once (a.k.a. SIMD). "CUDA by Example" first talks about translating traditional code into kernel functions - small functions that transform some given data only - that operate on the device. Afterwards, concepts and tips are given to help the reader utilise more of the GPU's processing power by introducing the grid-thread-block model, as shown in Figure 10. This abstraction shows how the CUDA runtime divides and distributes work across the inner processors on the graphics device.

Each thread and block will have related positional indices that identify it within its parent container. This unique numerical identifier can be decomposed into N-dimensional coordinates (usually two or three dimensional) to perform arithmetic on a specific chunk of the data. Partitioning the data in this way allows for the vast parallelism as the programmer could submit an entire block

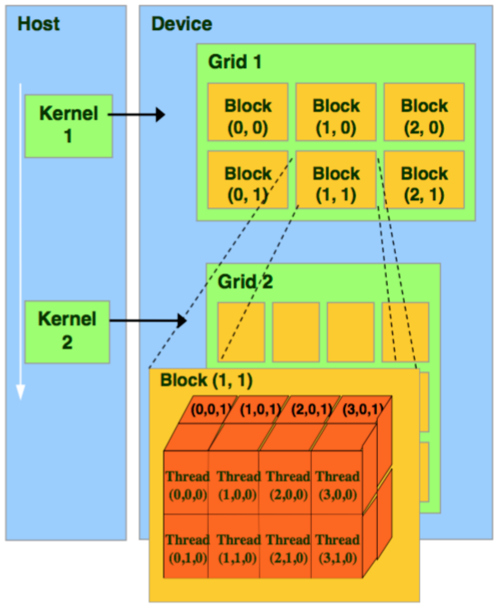

Fig. 10. Thread block abstraction model used in CUDA programming

or a small group of threads to the device for processing - there is a high level of fidelity granted to both data processing and transfer this way.

Before particulars of the CUDA technology are discussed, it's important to note that code that is written for CUDA must be exclusively compiled by the `nvcc` compiler provided in the CUDA Toolkit. This is the only way that CUDA-C code can get compiled into proper device code. Every other piece of non-CUDA code is ignored by nvcc, instead defaulting to the underlying C++ compiler available on the system - careful consideration of this is needed when attempting to use CUDA in a C-based project. General-purpose CPU code for this project needs to be compiled by the standard C compiler that is a part of the distribution being developed on. This could present issues when trying to use several specialised compilers in tandem to provide a solution that utilises multiple technologies together.

1) Parallelisation and Synchronisation: CUDA at its core is a library for writing very high concurrency code. This immediately brings about problems of how to correct establish and execute the parallel workloads, as well as how to effectively synchronise the tasks to ensure there are no race conditions, overlapping and incorrect memory reads, and correct numerical calculation.

The most powerful concept of CUDA is its use of kernel functions. These are regular subroutines affixed with the `__global__` keyword. This denotes that they are to be executed on the graphics device. Code decorated with the `__device__` keyword also only runs on device, and can only be called from `__global__` and `__device__` adorned code. This keyword is usually reserved for class-based constructs and supporting device code and algorithms. This kernel function gets executed on each CUDA thread, which each thread being a part of a block which in itself is within a grid (refer to Figure 10). Each grid and block have associated dimension vectors accessed via `gridDim` and `blockDim` respectively. Each block and thread have N-dimensional index vectors representing their position in their parent object, accessed with `blockIdx` and `threadIdx`.

It's a common pattern to divide the execution of a kernel function N ways for fine-grained control of parallelism. The thread id of the individual executing kernel function in two-dimensional space is defined by the following algorithm:

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

With this calculated, in order to safely perform work, the programmer needs to check that `tid < N`. If so, it's safe to execute the body of the kernel function as it won't cause any out-of-bounds exceptions or similar catastrophic errors.

Often multi-threaded functions will reach a point where all of the threads need to meet at a common instruction to move forwards. This is often the case in map-reduce algorithms where it's important that all mapping functions have finished on every piece of partitioned data before the reduce stage can commence.

For each thread in a block, it is possible to make faster threads hold for slower ones for full synchronisation by inserting the instruction `__syncthreads()`, an in-built CUDA method, wherever it is possible for every thread to call it. It's vital every thread calls this is or else the program will hang on some threads waiting for others to synchronise when they never will.

2) Streams: Parallelism has a shortfall (or advantage, depending on context) that interleaved, asynchronous operations that execute non-sequentially have no guarantee to execute in any imposed order. Any permutation of all possible interleavings is possible.

Streams are powerful CUDA constructs that allow for managed, controlled parallelism of CUDA functions. Having two streams will allow the same process to occur in down-times between different stages of execution of the same program. For instance, it will be possible for one stream to copy all of its required data into memory buffers and then execute its kernel function. At the same time as the first stream is executing its kernel function, stream 2 can be transferring its data into memory buffers as the data transfer bus is no longer being used by stream 1. This is inherently more useful as previously there would be a bottleneck of memory bandwidth with both memory transfers occurring at the same time.

In the section "Using Multiple CUDA Streams Effectively" in "CUDA by Example" [25], the authors recommend, in order to properly schedule work for the GPU in a way that is non-blocking and works according to the flow and inherent premises of the GPU's scheduling algorithm, that like-for-like actions should be done one-after-the-other across all streams instead of by blocks of operations. This leads to a waterfall or cascading effect of operations across all streams. Overall, this method leads to higher performance as the GPU can batch work together and perform several similar operations in parallel.

Concrete understanding and experimentation with this technology could lead to significant, orchestrated performance gains in SNAP also, if utilised correctly. If implemented incorrectly, it could easily bog the execution down. It may be the case that a concept as powerful as streams needs to be deeply integrated into the algorithm's design when the code base is young, or even from its inception, to have maximal effect.

3) Memory: Shared, Constant, Texture: Being largely used for rendering purposes for most of its existence means that the typical graphics card architecture comes with a host of different banks and types of memory and cache that accelerate and speed-up various graphics pipelines and calculations.

Shared memory is one of the most basic forms of memory available in CUDA. Shared memory is named as such as it is a special type of memory that is shared across all threads in a block, but not across blocks. This allows for a form of communication between threads. The memory chips implementing shared memory reside physically on the GPU, so are on-chip rather than off-chip. This proximity to its use allows for conservation of memory bandwidth and low-latency access times.

Shared memory is simply implemented by decorating a variable of choice with the `__shared__` keyword, as in:

```
__shared__ float cache[N];
```

Another performance-improving memory type that could be exploited for this project is *constant memory*. In graphics pipeline, several different processing algorithms, threads, and arithmetic logic units (ALUs) require access to the same scene. Often the frame of the scene is processed and the data and elements within the actual modelled world space can be considered to be unmoving and hence static or constant. To prevent a bottleneck of data, the information required by many different operations can be stored in constant memory on the graphics device. This memory is cached for conservation of memory bandwidth and is treated differently to that of standard memory on the graphics card. Nvidia typically places 64KB of constant memory on graphics card - including the GeForce GTX 1070 card being utilised on the development machine[9] - so it is very much similar to the various levels of cache chips found on a traditional processor rather than being akin to RAM. This size of memory will have close proximity and

[9]Information accurate at of time of writing

high-bandwidth access to the ALUs on the graphics board. Constant memory is guaranteed by the CUDA runtime to not change for the lifetime of the execution of a kernel function. The one trade-off is this data becomes read-only, so careful management is needs to circumvent this.

Placing data in constant memory is done by prepending the keyword `__constant__` to the variable in question. Assignment is done separately however. You must `malloc` or otherwise assign the data using standard C conventions and then call `cudaMemcpyToSymbol` to copy the data into the `__constant__` buffer denoted earlier. An invocation would look like this:

```
cudaMemcpyToSymbol(mem, temp_mem,
    sizeof(Data) * M);
```

Where `mem` is the constant memory, `temp_mem` is the standard assigned variable, `Data` is a data type used for each element, and `M` is the total size of the collection of `Data`. It is important to note the restriction that `sizeof(Data) * M` must be $\leq$ `64KB`.

Texture memory is a staple memory in graphical processing. It is an on-chip, cached, read-only back of memory that stores texture (bitmap graphics) data used by the texture units on modern graphics devices for sampling and transforming textures for application onto the surfaces of 3D models, as well as for lighting calculations. A particular property that texture memory has that constant memory does not is that of spatial locality of the data. Sampling a part of a texture often requires other threads accessing parts of the neighbouring data for their calculations. Figure VII-B shows this access pattern.

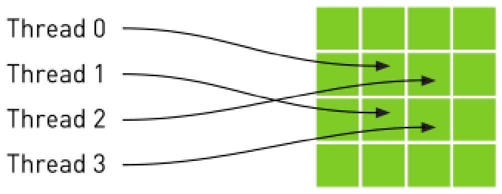

Fig. 11. Thread-based access of texture memory

Naturally, the access procedures and architecture of the memory chips and procedures are designed to use and exploit this. Hence, for two-dimensional data where the value of a cell might rely upon or otherwise update the values of neighbouring cells, utilising this type of memory would gain larger performance increases.

Textures are implemented by the generic `texture<?>` class, where `?` is the type of data being stored. Data that requires transfer into the texture, now bound in texture memory, must be allocated on the device, so requires a `cudaMalloc` call. Transferring data into texture memory from the allocated memory is accomplished with the `cudaBindTexture` or `cudaBindTexture2D` (for two-dimensional textures) functions. The only difference between the two is the amount of adjacent neighbours and the texture's layout in memory [26].

Textures must be unbound after use with `cudaUnbindTexture`.

Careful consideration of the use-cases of each type of memory for certain applications, especially ones that could usefully pertain to this project, will be a key consideration during the design stage as to which solutions are feasible and which aren't.

4) Multi-GPU Support: Finally, CUDA has built-in support for the utilisation of multiple graphics devices for a single program. Most high-performance computing clusters will use one or more graphics devices per node in their multi-node clusters, especially for modern applications.

The host operating system will encompass all the collective nodes and resources in them under one umbrella to make the system look like one computer with the aggregated CPUs, GPUs, RAM, and other resources together, all unified. CUDA is able to discern the properties of devices running on the host computer. Calling these functions with multiple graphics devices will return a list of devices and their capabilities, each with their own ID and set of properties. In a homogeneous computing cluster, each node will have the same resources as thus there will be no difference as to what part of the program will be executed on which. However, in a heterogeneous computing environment, it will be advantageous to interrogate each device and allocate appropriate work to appropriate devices. For example, if a workload can only be split up into one large subtask and several supporting ones, a feasible assignment would be to put the fastest or largest device to the task of the large subtask and to distribute the smaller subtasks randomly across the remaining devices. It will all be down to the resources allocated to the submitted job or whatever is available.

Using `cudaSetDevice(id)` allows for targeting of a thread's kernel function to a particular device. The rest of the function and clean-up procedure remain the same. Knowledge of devices is very proprietary and optimisations may be very specific. A viable design goal worth exploring could be to implement a smart device querying and work scheduling/partitioning system during the initialisation of the application. However, there is no guarantee that this generalised approach would yield significant advantages.

IV. Design

Principally, the aim of this project is to establish, as a proof-of-concept, whether an existing code base can be meaningfully accelerated using CUDA technologies. A complete rewrite is not the goal as SNAP is a legacy project.

Establishing the efficacy of the proposed solution will be done against a best-case scenario model of parallel execution. Knowing this model is not attainable in practice for this project is acceptable (and not in scope). It will serve as a solid basis for future work should a complete refactor or related work be undertaken.

Following the conceptual, pure model, potential designs are remarked upon with their advantages and disadvantages highlighted, and supporting programs specified afterwards.

A. Mathematical Model

In order to understand the amount of reasonable work the GPU could handle at once, as well as estimates for the amount of time and data movement needed for the optimal implementation, a mathematical model was developed to help visualise and describe the interactions between adjacent cells in the approximated algorithm that SNAP uses. The following assumptions were made when developing this model:

1) The algorithm defines the problem space as a three-dimensionally discretised grid of cells, each with their own values and interactions with their neighbouring adjacent cells. This forms the core of the SNAP algorithm (it itself approximating a continuous algorithm).
2) The problem space is cubic with side n for ease of modelling. A generalisation to a cuboidal space with dimensions (x, y, z) can be abstracted from the cubic model at a later time. A cubic model is conceptually and mathematically more intuitive to model.

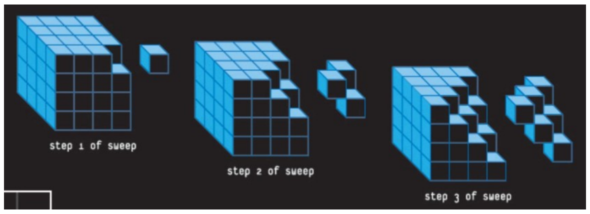

Fig. 12. Slices of a sweep of 3D grid

Given these assumptions, we aim to derive the total number of communications between cells in the grid per iteration and how much work this equates to and how much can be parallelised at each stage.

There are 8 vertices of a cube so 8 octants to perform the sweep over. Each sweep is divided up into diagonal slices, shown in Figure 12, of which there are $(x + y + z) - 1$ of them ($3n - 1$ in this model). A key point to note is the dependency in each sweep of a slice on the slice before as the direction of the sweep is linear and calculations rely upon previous ones[10], and each combined sweep of all octants, after all results being collated, forms the basis of the next sweep.

The first n slices of the grid are comprised of the sum of the first n triangular numbers' (i.e. $1, 3, 6, 10, 15\dots$)[11], worth of cells, with the final n slices having the same number also (just in reverse order). $n - 2$ slices exist between the 2 triangular-based pyramids formed from the previous step, allowing us to develop a function for the first $m = n + \frac{n-2}{2}$ slices, and mirror it to get the number of cells in any slice of the grid.

The number of cells in slice i of a cube of size n is thus given by the following formula:

$$cells(i) = \begin{cases} tri(i) & \text{if } i \le n \\ tri(i) - 3tri(i-n) & \text{if } i \le m \\ cells(m - ((i-1) \bmod m)) & \text{otherwise} \end{cases} \tag{1}$$

Where $tri(i)$ represents the ith triangular number, $1 \le i \le 3n - 1$.

This means poor performance early on due to low numbers of cells per layer and the inter-slice dependency, but will ultimately scale well and to more graphics card given a large grid.

To work out data transfer, we need to figure out the amount of sends and receive actions are performed per slice. We can again work first out up to the first m slices, and take the mirror image for the rest of the slices, swapping the number of sends per slice for the receives of the mirror slice and vice versa. For the first $n - 1$ slices, each cell sends to 3 neighbouring adjacent cells in the next slice. When $i \ge n$, we need to take into account the "cut-off" portions where some cells only send to 2 neighbours, hence:

$$sends(i) = \begin{cases} tri(i) * 3 & \text{if } i < n \\ (i \bmod (n-1)) * 2 + & \\ (cells(i) - i \bmod (n-1)) * 3 & \text{if } i \le m \\ recvs(2n - i) & \text{otherwise} \end{cases} \tag{2}$$

The total number of receive actions is similar. The first cell has no neighbours to receive from, the second slice has only 1, the rest have three, then after the nth slice we need to consider where we only receive from 2 neighbours:

$$recvs(i) = \begin{cases} 0 & \text{if } i = 1 \\ tri(i) & \text{if } i = 2 \\ tri(i) * 3 & \text{if } i \le n \\ (cells(i) - (2n - i)) * 3 & \text{if } i \le m \\ sends(2n - i)\text{otherwise} & \end{cases} \tag{3}$$

Both functions are defined in the domain $1 \le i \le 3n-1$.

If we let msg_s be the size of a message to send and msg_r the size of a message to receive from a neighbour (both in number of bytes), and knowing that the fastest data transfer rate of the PCI Express (PCIe) 2.0 bus connecting the graphics card and CPU together is 500MB/s [27], we can calculate the time per slice to store data from the GPU as

$$time_s = \frac{sends(i) * msg_s}{5 \times 10^8} \tag{4}$$

[10]As described in the documentation of the SNAP algorithm in their GitHub repository - https://github.com/lanl/SNAP/blob/master/docs/SNAP-overview-presentation-2015.pdf

[11]Sequence A000217 in the "On-line Encyclopedia of Integer Sequences" (https://oeis.org/A000217)

and to send data to it as

$$time_r = \frac{recvs(i) * msg_r}{5 \times 10^8} \tag{5}$$

Aggregating these over all slices, alongside an assumed small, constant kernel function calculation time C over $cells(i)$ cells in the slices, yields the best-case calculation time for the current approach. From here, we can interleave all 8 octants for calculation at once (of course managing the number of streaming multi-processors, which, after a point, we'd need to vastly scale hardware or queue work) to improve our throughput. C ultimately depends on the type of graphics card being used.

B. *Ideation*

- In-place replacement
- Loop flattening

C. *Test Harness*

In order to accelerate the process of testing, analysis, and comparison of data and timings yielded from the runs of the two separate SNAP variants, a decision was made to create a test harness program that will automate the process of running and collecting results.

Automation with regards to testing is key to repeatability, reproducibility, the speed at which results can be obtained, and general accuracy of the results. Hence, for testing of both programs to be effective, the test harness program must meet certain criteria in a holistic process to ensure the project is a success. The overall aims of the program are: to act as a wrapper to run both versions of the program with; to provide high-precision timing of executions; to handle any errors thrown; and to output, compare, and save the results of executions for analysis and reporting.

Firstly, the program should act as a wrapper to the execution of both programs. Manual execution of the programs is slow, can be inaccurate, parameters can be missed accidentally, and executing a sequential string of commands, each varying a little (e.g. only changing the input and output destinations) is tedious. Programmatic control will expedite the testing process and provide extra flexibility should the testing process need to be enhanced or altered in the future.

Further to this is, crucially, accurately timing the execution of both programs. It would be an ineffective solution to the problem if the GPU-accelerated version of SNAP would be routinely slower than the original. Specifically, timing both the entire execution of the program and the internal algorithm/function we're seeking to modify would give an overall impression of the efficacy of the solution. For example, if the algorithm's execution time has decreased but the overall program's execution time has increased due to knock-on factors as a result, then the solution would need further work.

Advanced control could not be achieved if the harness were not able to capture output and errors of the underlying programs that it was calling. It's important to know when errors are found as it might be necessary to halt the problem and report it or continue on regardless depending on what happened - the solution needs to provide this support and subsequently handle it. To this end, obtained the output of both of the programs is a required detail to analyse problems, but most importantly, to compare the final numerical results of the execution on identical inputs to ensure that the results are the same. Inconsistent results mean an improper, incorrect solution. This comparison and analytical process needs to also be in part to mostly automated to save users from wasting time doing it manually and potentially not spotting a discrepancy or difference.

A program that meets these central requirements will act as an effective and useful testing tool for assessing if the end solution meets the objectives of this project.

V. Implementation

An experimental alteration to an existing code base can often be wrought with challenges. Therefore, a solution to this problem that meets all the objectives outlined earlier must satisfy the following criteria:

- **Accuracy** - Simply put: if the results of the program after the modification do not align with the results of the unmodified program, then the solution is not satisfactory and is thus incorrect. Care and constant ratification must be performed often to ensure this trait is maintained as development occurs.
- **Speed** - A solution that is not better than or comparable to the existing program's performance is worse in practice. In the worst case, the program should be at least as fast.
- **Simplicity** - A solution that is conceptually harder than the existing code is not only not as maintainable but increases the fragility of the algorithm. Code that is improved *and* intuitive can lead to further improvements down the line such as performance gains, reduction of necessary code, or easier parallelisation.

Accuracy and speed are easily assessed using simple comparative automations and time-keeping. These were the key motivators for introducing a test harness program into the development and testing cycle (see Subsection V-B for development details). Simplicity is a more subjective metric for evaluating a software-based project. A best-attempt at an objective review of the implementation will be featured in the conclusion (Section VII). However, personal preference and opinion will always factor into such a review, so the reader can also review the code base provided to make up their own mind.

Development was undertaken iteratively after installation of all of the necessary prerequisites to start programming. The basic cycle mimics aspects of the Agile development cycle[12] and other traditional development cycles:

1) Partition the current problem into a list of remaining sub-problems.
2) Write code to solve the current sub-problem.
3) Test the code.
    a) If it works, move forward to step 1.
    b) Else, go back to step 2.

This project diverges from this slightly in that during the second stage code was written for both the CUDA-based alteration and the test harness. As one was developed, the other was developed and tweaked further. Both were kept in step to ensure accuracy and speed. Implementation details of component of the project follow.

A. Prerequisites

To begin development, in addition to the installation of the necessary Ubuntu packages for C/C++ compilation and python development, it's necessary to install binaries associated with the CUDA development kit and runtime.

A proprietary device driver for Nvidia graphics cards on Ubuntu is required for CUDA code to run on the device. This is provided as part of the "CUDA Toolkit" that is freely available to all developers from Nvidia's Developer portal https://developer.nvidia.com/cuda-downloads. Installation is a matter of following the steps in the installer. It was important to note that the driver provided with the installer is the one guaranteed to work with the CUDA Toolkit, as opposed to the ones provided as part of the operating system vendor's software package ecosystem. Modification of the `PATH` and `LD_LIBRARY_PATH` to reference the `/usr/local/cuda-10.1/*`[13] folders sets up the running environment properly and, with careful compilation, now allows for compilation and execution of CUDA code via the `nvcc` compiler.

The operating graphics driver version provided with the CUDA Toolkit download was version 418.67. It had to be made absolutely imperative that this was the running version of the driver, as on Ubuntu there are multiple different versions one can install. The ones on the development machine were nvidia-driver-418, nvidia-driver-390, and an open-source X.Org X server display driver that the distribution defaults to. Sometimes on boot the system may not be able to load 418 for whatever reason so there are some instances where a reconfiguration under Software & Updates → Additional Drivers and a restart required to get the correct driver running. At this time, it is unknown whether this issue is caused by a bug in the operating system or the underlying hardware. An incorrect running driver will either cause any attempt at running CUDA code to halt because the compiled device code cannot properly access the graphics device (the main job of the driver) or cause aberrant, incorrect results during calculations on the device.

One last prerequisite is to have the Intel MPI libraries installed an operational on the development machine also. This is covered in Subsubsection III-B1.

B. Test Harness

Developed in conjunction with the modified SNAP application (Subsection V-C), the test harness program is a versatile complementary program that aims to help achieve the objectives of the project by providing the following features:

- Acts as a wrapper to both CPU and GPU-augmented versions of the SNAP programs.
- Precisely times their execution.
- Compares outputs of the programs to gauge consistency.

For this supplementary program, Python version 3.7+ will be the main implementation language. The core libraries of Python, in conjunction with its ease of development and portability will allow the program to be developed rapidly and be more intuitive. Version 3.7.3 is used explicitly due to its modernity (it was the latest available distribution at the time of writing) and this script's reliance on the `time.time_ns()` function (the use of which is described later).

Three of the main APIs utilised in the program that meet most of the needs of the above requires are the `os`, `subprocess`, and `time` libraries found in Python's standard set of core libraries. These libraries provide tight, low-level, flexible interfaces for working with the host operating system in addition to providing high-precision timing facilities for program comparison and profiling. Further advantages are discussed as the implementation details of each point are outlined in turn.

The primary goal of the test harness program is to act as a wrapper to execute either one of the SNAP programs. This control allows for a fine-level of timing and parameter control that isn't guaranteed with manual execution. Python's `subprocess` library can interoperate with Ubuntu and execute shell commands specified by strings in the script and have them execute within a subprocess. Not only does this provide exact start and end times of when the underlying subprocess is forked and joined back into the main executing process, but the library provides an extensive API to handle and report errors, output, and stream input for a variety of sources, amongst other features. Specifically, `subprocess.call([...])` is liberally utilised as it takes in an array of parameters (the head of which being the program binary name) and executes it as if it were a shell command. Programmatically inserting input parameters, the path of the SNAP binary - chosen by an argument passed into the `test.py` script itself - to execute, as well as any additional information and arguments for manipulating the execution or profiling

[12]https://www.agilealliance.org/agile101/

[13]The latest CUDA Toolkit version at time of development.

characteristics (see Subsubsection V-B1 for further details). If it is ever the case that written algorithms in the script could be deferred to an existing piece of software or bash script (e.g. `diff` or `cmp`) then this could also be used to refactor the program to utilise those instead.

The `os` library handles difference between operating systems types to make the program as OS-agnostic as possible. Details such as file path separators, reading, and writing are handled through a common API. This allows for easy reading of an input or output file into the program for line-by-line parsing by several other routines and libraries used for the other stages and parts of the script. `os.remove` is used to delete the extraneous `gmon.out` files produced by profiling, while `os.path.join` is used to link scripted parts of paths together to find the output directory and particular files within it for averaging of execution times and comparison of the contents of the files - alongside the `open(...)` function from standard Python.

High-precision timing is provided by the `time` library: specifically, the `time.time_ns()` function. This function was introduced in Python 3.7 when nanoseconds resolution was introduced to the `time` library[14]. Precision and accuracy is key when doing time-based profiling specifically, hence the require for the nanosecond time resolution. `time` previously only had the `time.time()` function which only returned the number of seconds since a certain date, i.e. Unix epoch time, and this was certainly not appropriate.

Detailed breakdowns via the use of `MPI_Wtime()` to time the start and end times of the `dim3_sweep` function and its parent subroutines in the call stack were experimented with. The cost of the intricate implementation of summing all the values from all of the MPI processes outweighed the benefit of the insight it would provide so this was not included in the final build. Use of `gprof` and other profilers provide similar functionality already.

Output from the file is saved, immediately after execution, in a file with the following naming convention: `outNN_cpu_n4_1`, where

- *NN* is the two-digit number corresponding to the respective input file from the `qasnap` directory.
- *cpu* is interchangeable with "gpu" depending upon the particular application that was run.
- *n4* denotes the number of MPI processes specified as a parameter.
- *1* denotes the id of the run. Subsequent runs of the same configuration will increment this value.

This nomenclature uniquely identified the executions and allows for rapid processing during analysis. All of the output files are stored in a separate directory called `out/` (which is automatically created if it doesn't exist).

[14]https://docs.python.org/3.7/whatsnew/3.7.html#pep-564-add-new-time-functions-with-nanosecond-resolution

Post-processing of the output file generated from the SNAP program is done immediately after execution provided that no error was reported. If the program returned a non-zero exit code, then the program reports this and cleans up immediately afterwards. Post-processing involves stripping the output of the time stamp at which the execution occurred because this is unnecessary information that gets flagged up by comparison programs and algorithms, and splitting of the output file into two halves. The first half contains only the values associated with the numerical calculation and simulation, and the second half, which gets stripped from the original output file and written to a new file of the same name except with the suffix `.timing`, contains the subroutine and overall times of the execution. This bifurcation was done to simplify comparisons to only the like-for-like data. Numerical values output by the software can instantly be compared (give or take minor affordance in terms of rounding and such) to gauge the accuracy of the solution, whilst the separate timing information can be utilised in a separate program (or manually) to garner an appreciation of the difference in performance between the two programs.

Automatic creation of the output file and its name is a key advantage of the harness program's automations. Using a standardised name and processing procedure allows for substantially easier further processing, validation, and parsing.

Readily available tools for identifying the similarity between two files is axiomatically needed as manual comparison of the output files the SNAP application gives are detailed. Any inconsistencies or numerical changes might be hard to spot.

A `compare` command was added to the API of the test harness as a post-execution analysis tool to perform a line-by-line comparison of the two input files specified. The calculation in the final version presents a simplified, line-by-line comparison algorithm that count the amount of lines that differ and offers this as a percentage as its output. 0% change is identical, any percentage difference under 10% is near-identical, and any other percentage is different.

If the user specifies the `-v` option with the `compare` tool, the application will call the tried-and-tested `diff` application that is bundled as standard as part of most Linux distributions. This invocation to `diff` uses the `-s` and `-y` parameters to show a side-by-side comparison with the differences between the two highlighted, and gets the program itself to verify whether or not the files are identical as a verification to the test harness's calculation.

Tabulation of timing data for comparison later on during the analysis process needs to take into account variations between runs. The `average` command was introduced to average the overall time taken per group of related output files (the relation being that the output files all have the same configuration). The average time is simply extracted from the populated `testing.csv` data file that the `test` program maintains and appends to after each invocation of the SNAP binary through its run command, and then divided by the number of runs that particular configuration has been executed.

This averaged value is far more representative of the overall time taken for the program with given input and parameters. This value is what will be used in the final testing. If more fine-grained comparison of the timing data is needed, `test compare` can be invoked on two `.timing` files to see the differences.

1) `test.py` API: The test harness assumes the form of a program operated solely via the command-line. As such, there is a small set of commands and arguments associated with the various functionalities implemented and described in the design. A reference follows, all of which can be accessed via a terminal by issuing the command `python test.py -h`.

- **setup**: Establishes the executing environment if not already configured.
- **clean**: Cleans any superfluous or unnecessary files from profiling or previous runs that are otherwise cluttering up the working directory.
- **run**: Executes one of the SNAP binaries in the project using the following arguments:
  - **–cpu**: Makes the program choose the original binary file of the SNAP program. Must define either this flag or **–gpu**.
  - **–gpu**: Makes the program choose the modified, GPU-accelerated binary file of the SNAP program. Must define either this flag or **–cpu**.
  - **-n, –np**: Number of MPI processes to run with.
  - **-i, –fi**: Path to the input file to test the program with.
  - **–clean**: Removes profiling data and extraneous information and files from previous runs.
- **average**: Calculates the average time of executions meeting a particular configuration.
  - **-i**: Name (not path) of the input file used that was passed into the SNAP application.
  - **-n, –np**: Number of processes the execution ran with
  - **–cpu**: Whether the execution was run with the original version of SNAP. Mutually-exclusive option with **–gpu**.
  - **–gpu**: Whether the execution was run with the modified version of SNAP. Mutually-exclusive option with **–cpu**.
- **compare**: Performs a line-by-line comparison of two input output files from previous runs. Return a percentage difference of the two. Any close to 0% are reported as “identical”, less than 10% “near-identical”, and anything else is “different”.
  - **–f1**: Name (not path) of output file to compare.
  - **–f2**: Name (not path) of second output file to compare to first.

The following global flags are also available for any command:

- **-v, –verbose**: Increase the amount of output of the program for diagnostics.

C. Modified SNAP Application

Developing the GPU-accelerations for SNAP first involved understanding how to integrate the CUDA algorithms into the existing MPI-based C implementation.

It was required to get both the `nvcc` and `mpiicc` to co-operate with one another, first and foremost.

Secondly, in terms of the process of development, was to attain an understanding of how best to integrate CUDA and MPI code together. An issue presented immediately is that of both the specialised compilers splitting the compilation process into stages. Each compiler will compile separately the specialised parts of the code that the standard C/C++ compiler does not know how to handle, whilst, either at the same time or after the first stage, it will call the standard, underlying compile to compile the rest of the source files. Afterwards, the specialised compile will link both compile non-standard and standard object files together into one executable binary file. Having both compilers do this process means that specialised MPI and CUDA code may not be able to easily co-exist in the same source code files without separation or careful orchestration of the build process.

1) `nvcc` and `mpiicc`: As discussed earlier, `nvcc` and `mpiicc` are both non-standard code specific compilers that defer to the standard C/C++ compiler once the specialised code they target have been compiled into object code. An issue lies in which compiler either one looks for: `nvcc` looks for the standard C++ compiler, `mpiicc` looks for the standard C compiler. This discrepancy caused an issue when trying to link the compiled object files of both compilers together.

During the investigation, the provided Makefile was complete and able to compile the existing project. A few modifications were made to the parameters passed to `mpiicc`. The full list ended up being `-mkl -O3 -std=c11 -xAVX -fopenmp -parallel -DUSEMKL -DUSEVML -pg -g`. The targeted C version (`-std`) was updated to c11, `-pg` was added to allow `gprof` to profile the code, and `-g` to allow for increased debugging.

For the new `.cu` files, two new variables were added to define the compiler and flags for these file-types:

```
GPUCC=nvcc
GPUFLAGS=-arch=compute_61 -code=sm_61 -m 64
    -rdc=true -dc -dlink
```

`nvcc` is the compiler, and the meaning of each of the parameters is:

- *-arch=compute_61* - Specifically targets the GTX 1070 and related device architecture.
- *-code=sm_61* - Defines architecture to optimise and generate device code for, in this case the GTX 1070 and related devices.
- *-m 64* - Targets 64-bit architecture.
- *-rdc=true* - Enables generation of relocatable device code.
- *-dc* - Compiles each `.c` and `.cu` that contains any relocatable device code.
- *-dlink* - Links object files with relocatable device code into executable device code.

In addition to the following compilation rule that simply takes `.cu` files as dependencies and compiles them, using the above variables, into `.o` object files:

```
%.o: %.cu
    $(GPUCC) $(GPUFLAGS) -c $< -o $@
```

Separate `.cu` files are used to separate and easily compile CUDA code from existing MPI code (this is discussed further in the next section).

Linking these files together involves a new compilation rule, shown in Figure 1reffig:nvccmpiicc, using `nvcc` as the linking compiler as it is best suited to integrating the more non-standard CUDA code (MPI code is closer to standard C). It calls `mpiicc` as the backup compiler, which, in turn, will defer to the standard C compiler on the machine.

```
$(A_TARGET): $(A_OBJS)
    $(GPUCC) -arch=compute_61 -code=sm_61 -
        m 64 -std=c++11 -ccbin=$(CC) -
        Xcompiler "-mkl" $(LDFLAGS) $(LIBS)
        $^ -o $@
```

Fig. 13. New make rule for linking MPI and CUDA

The latest C++ variant is used, alongside targeting of the 64-bit architecture and the GTX 1070-series (and family) graphics device architecture. New here is the inclusion of `-ccbin` which points to `mpiicc` as the compiler to refer to. In order to correctly compile with this sub-compiler, the `-Xcompiler` flag is needed to build with the Intel MKL library. Fortunately, this is the only thing that's needed to correctly use this. Relevant linker flags for non-standard linking libraries (`-L/usr/local/cuda/lib64`) as well as CUDA linking variables (`-lcuda -lcudart`) are used to link all of the object files together into an executable binary. Taken together, this successfully produces a working binary that utilises both CUDA and MPI and OpenMP.

2) Externalised CUDA Algorithms:

VI. Testing

Verification of the modified software will take place in two stages on different sized platforms, as outlined in subsubsections II-C1 (Local Development Platform) and II-C2 (Computing Clusters) so see if the code scales. The primary objective is to attempt to observe a time-decrease over every problem size in the best case scenario. Other objectives are:

- Ensuring the modified solution produces the same results as the original version is all circumstances.
- A subsequent secondary performance increase is observed in multi-GPU environments. This is supplementary and not required, as CUDA has particular technologies associated with distributed work across multiple devices.

There are a couple of main variables that would immediately effect the performance of both programs: the number of CPU threads, the amount of MPI processes, and the block and thread configurations of all CUDA-based kernel function calls. Several variations of these will be incorporated into the testing as they will show trends in parallelism that might not otherwise be observed if just a few fixed values are used. For instance, it may be the case that the algorithm doesn't scale well to higher numbers of processes or threads.

A. Test Plan

To meet these objectives, the following test plan is laid out. Cluster-based testing may be omitted depending upon the results obtained during local testing - it may be the case that there is no apparent performance increase.

Starting with local testing:

Input File	Procs
in01	4
in02	6
in03	6
in04	4
in05	4
in06	4
in07	8
in08	8
in09	2
in10	4
in11	4
in12	4
in13	4
in14	1

Once, and only if, all of the testing has passed up until this point, we can proceed with cluster-based testing:

Value 1	Value 2
1	2

B. *Local Testing Results*

C. *Cluster Testing Results*

D. *Results*

VII. Conclusion

A. *Limitations*

- Very proprietary code base

B. *Improvements*

Over the course of the project, several issues or development quirks came to light in almost every aspect of the pipeline. Several key improvements that could be made to the project are presented below that could rectify these should subsequent or related projects take on a similar endeavour to the one presented in this report:

- **Texture memory**: One core feature in the CUDA library that wasn't explored due to learning the legacy code base, pursuing alternative designs, and other time constraints, is texture memory and the potential solution and performance benefits it could bring. Texture memory is specialised, cached memory that improves the memory bandwidth of a running parallel application by exploiting the spatial locality of the data in question, as in Figure shown earlier.
  For restricted transport-sweep problems over two dimensions, texture memory for the implementation of the grid would greatly speed up reads from memory. Each thread can operate a different directional sweep across the same cached texture, synchronise all of the threads, and aggregate the results back before transferring data back to main memory and the CPU. This approach has promise and would be ideal to pursue further for problems like SNAP.
- **Containerisation**: An issue that appeared time and again during the initial investigation stages was the lack of other projects that had tied together MPI and CUDA as this project was seeking to do. Whilst not unusual for new and novel work, the lack of supporting documentation for this use-case proved to be a hinderance to time and slowed parts of the project down. A key piece of future work for this project in the future could be the establishment of a Docker container (a software housing that guarantees consistency of execution no matter what hardware it is run on) to hold binaries and settings for future release and other researchers to use the technologies together, alongside supporting documentation detailing lessons learned during the course of development.
- **Standardised input data format**: Modernising and documenting the input data structure would help to vastly improve the fidelity and flexibility of the program by allowing faster and more intuitive changes to be made without prior knowledge. As it stands, the input format is very proprietary and analysis of the code, particularly `input.c`, is needed in order to interpret the variables and what they mean in context. Utilising a modern, global data standard such as JSON or XML over a whitespace dependent ordering would increase the usefulness of input files used for testing. Not only are there high-performance, fast, compact parsing libraries available for these formats but they're also intrinsically easier to document and modify.
- ...

References

[1] M. Office, "Unified Model," 2018.

[2] L. Mindy Weisberger, "A New Supercomputer is the World's Fastest Brain-Mimicking Machine," 2018.

[3] S. Scoles, "This Bomb-Simulating US Supercomputer Broke a World Record," 2018.

[4] TOP500.org, "Top 10 Sites for November 2018," 2018.

[5] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[6] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, pp. 103–104, 2007.

[7] NVIDIA, "Cuda toolkit documentation," 2018.

[8] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pp. 836–838, IEEE, 2008.

[9] Y. Tang, "Deep learning using linear support vector machines," *arXiv preprint arXiv:1306.0239*, 2013.

[10] 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," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[11] J. A. Favorite, "A brief user's guide for PARTISN," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2011.

[12] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, vol. 2013, 2013.

[13] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "Gpu behavior on a large hpc cluster," in *European Conference on Parallel Processing*, pp. 680–689, Springer, 2013.

[14] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on gpus," *Computer Physics Communications*, vol. 192, pp. 97–107, 2015.

[15] Y. Zhou, J. Liepe, X. Sheng, M. P. Stumpf, and C. Barnes, "Gpu accelerated biochemical network simulation," *Bioinformatics*, vol. 27, no. 6, pp. 874–876, 2011.

[16] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, and W. Joubert, "High performance radiation transport simulations: preparing for titan," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 47, IEEE Computer Society Press, 2012.

[17] H. Shoukourian, T. Wilde, A. Auweter, and A. Bode, "Predicting the energy and power consumption of strong and weak scaling hpc applications," *Supercomputing frontiers and innovations*, vol. 1, no. 2, pp. 20–41, 2014.

[18] M. Martineau and S. McIntosh-Smith, "The productivity, portability and performance of openmp 4.5 for scientific applications targeting intel cpus, ibm cpus, and nvidia gpus," in *International Workshop on OpenMP*, pp. 185–200, Springer, 2017.

[19] O. B. Messer, E. DAzevedo, J. Hill, W. Joubert, S. Laosooksathit, and A. Tharrington, "Developing miniapps on modern platforms using multiple programming models," in *2015 IEEE International Conference on Cluster Computing*, pp. 753–759, IEEE, 2015.

[20] V. Driessen, "A successful git branching model," 2019.

[21] S. C. R. T. Platform, "High performance computing," 2019.

[22] Intel, "Intel® mpi library developer reference for linux* os," 2019.

[23] D. Kanter, "Intel's sandy bridge microarchitecture," 2010.
[24] "Intel® mpi library developer guide for linux* os." https://software.intel.com/en-us/mpi-developer-guide-linux-introducing-intel-mpi-library.
[25] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.
[26] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.
[27] "Pci express - wikipedia," 2019.