Integrating CUDA into SNAP

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 codebase 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 codebase and architecture, a thoughtout 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. 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

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

[2] https://www.openmp.org/

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

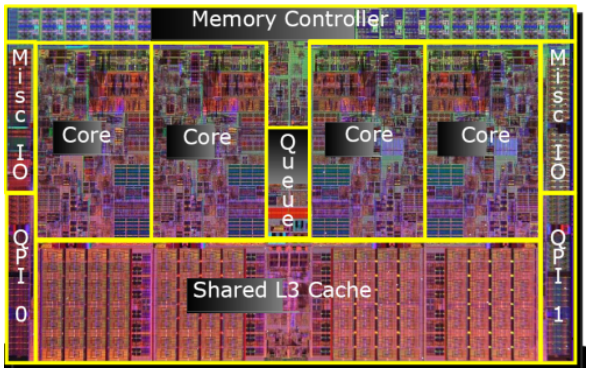

Fig. 1: Quad-core Intel Core i7 CPU Architecture Diagram

Fig. 2: Nvidia Turing GPU Streaming Multiprocessor Architecture Diagram

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 are architected differently to 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 ("miniapp") 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]. Miniapps 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].

B. Objectives

The SN (Discrete Ordinates) Application Proxy (SNAP) is a miniapp 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 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

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

With this miniapp 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 miniapps was written by Heroux et al [10], defining the paradigm. Their Mantevo miniapp suite has show successful development of miniapps, 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). Miniapps 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 codebases [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 miniapps 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. Mallison 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 miniapps 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 miniapps 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 miniapp must adhere to and the considerations of forming a miniapp 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 courseworks 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 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 unfortuitous circumstances arriving - whether

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

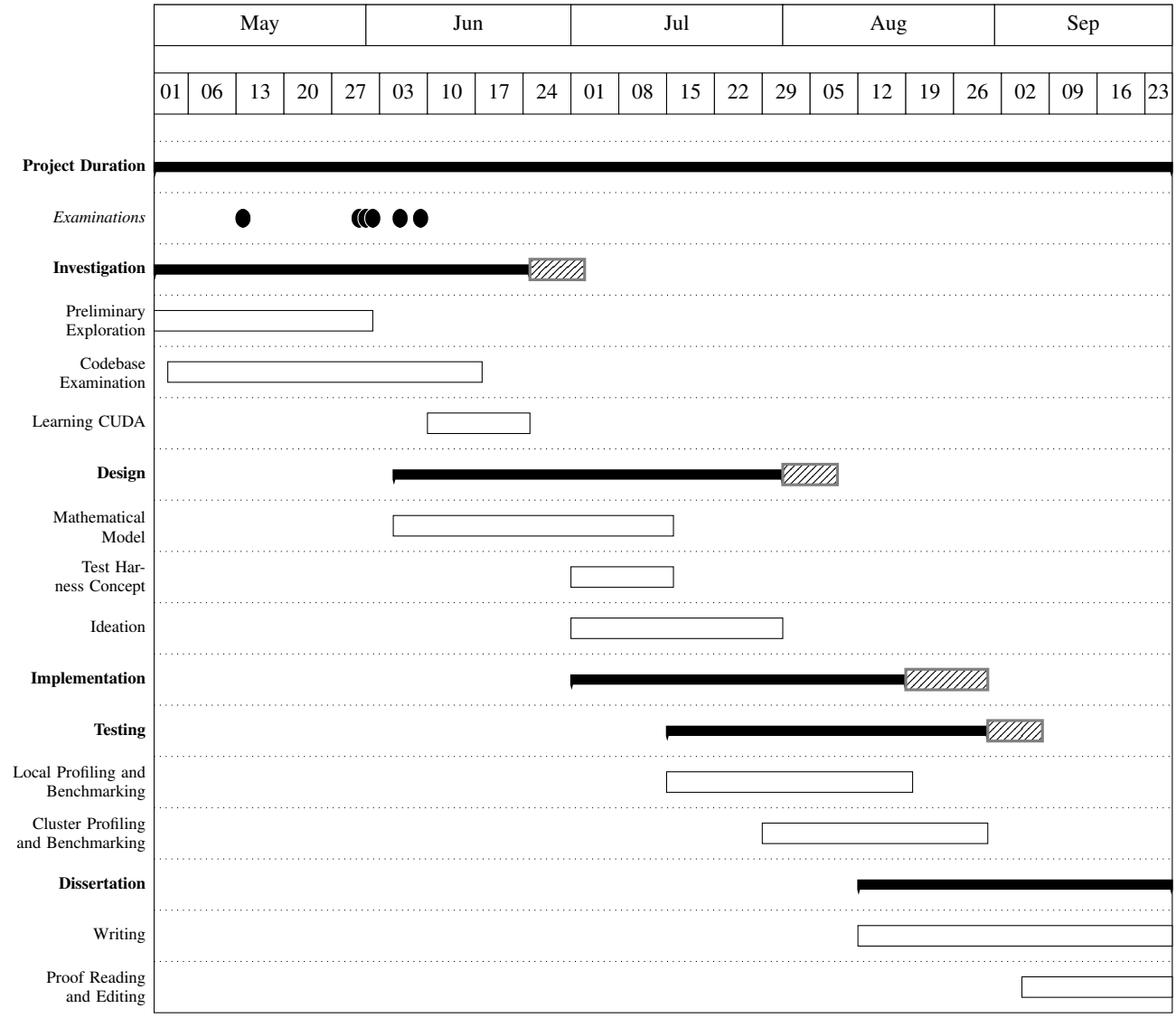

Fig. 3: The Proposed Timeline of this Project

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 codebase 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 demark stable versions when they are created.

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 miniapp also additional hardware requirements in an Nvidia-based GPU and accompanying drivers and development suites. If the modifications to the codebase 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 setups follow.

1) Local Testing: 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™ 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 hyperthreading 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 setup 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™ 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.

2) Cluster Testing: ...

3) Software: Working with an existing and established codebase means adhering to existing toolchains 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 encorporate 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**: Backend 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.

...

III. Investigation

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.

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 - specfically fused multiply-accumulate (FMA) operations - for the execution of some SIMD workloads on floating-point data [20]. 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[6]. `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 [21] to allow for bi-directional inter-process communication, regardless of how those processes are mapped to the underlying hardware.

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

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. 4: Important variables defined in the input data

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 assocaitions 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.

2) Understanding the Input Data: Input data in particular format

3) Execution: Following successful compilation of the SNAP executable, several executions were performed under various environmental-constraints and input data to ascertain the current limitations and performance of the codebase 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 6 demonstrates the output of a successful execution of a given input.

It was identified early on into the testing - whilst undocumented, it could be said to be a core feature of

```
#!/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

```
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. 6: A successful execution of the compiled snap_mkl_vml binary

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 7 shows the error message provided in this scenario.

C. gprof

Without knowing sight-on-scene which aspects of the codebase are called from which parent subroutine, for whatever piece of input or interim data, and what time, it's necessary to perform time-centric analysis to gauge how much time is being spent in which particular subroutines as these are the main bottlenecks when it comes to the performance of an algorithm.

1) Code Analysis:

D. Learning CUDA

Being the only way to usefully execute general-purpose code on a(n Nvidia) GPU, it is important to comprehensively understand CUDA before delving into making any modification to the source code of this project.

- Architecture
- Kernel code
- Shared memory, thread cooperation etc

IV. Design

A. Mathematical Model

In order to get a gist of the amount of reasonable work the GPU could handle at once and 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-dimensionality discretized 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 later.

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 9, 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[7], 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. $1, 3, 6, 10, 15 \ldots$)[8], 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:

[7]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

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

```
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=0x7ffd07f649b0,
     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. 7: An unsuccessful execution of the snap_mkl_vml binary due to imposed multi-processing limitations

% 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. 8: Amalgamated gprof log (truncated to top 5 rows)

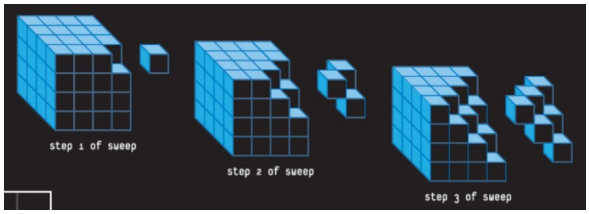

Fig. 9: Slices of a sweep of 3D grid

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

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

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 $i \geq n$, we need to take into account the "cut-off" portions where some cells only send to 2 neighbours, hence:

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

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:

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

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

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 [**?**], we can calculate the time per slice to store data from the GPU as

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

and to send data to it as

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

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

C. Test Harness

V. Implementation

A. Prerequisites

- Nvidia drivers
- CUDA libraries

B. Test Harness

- Python
- OS-calls and handling
-

C. Modified SNAP Application

- Intel C MPI library
- Externalised CUDA algorithms
- nvcc and mpiicc

VI. Testing

- Test Plan
- Test Harness Application
- Local Testing

VII. Conclusion

A. Limitations

B. Improvements

- Docker container for a consistent operating environment
- Modernise the input data into a new format (e.g. JSON)

C. Further Work

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] D. Kanter, "Intel's sandy bridge microarchitecture," 2010.

[21] "Intel® mpi library developer guide for linux* os." https://software.intel.com/en-us/mpi-developer-guide-linux-introducing-intel-mpi-library.