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 actually 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 in addition to offloading applicable workloads to graphical processing cores, aiming 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 related HPC disciplines and projects is presented The target miniapp SNAP is discussed in addition to outlining the objectives of what the project aims to achieve by modifying it. Current efforts and progress are detailed and reflected upon against the timeline offered in the project proposal previously submitted as part of this project/module. Finally, further work and a future timeline is described.

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 work, and each has multiple open-source implementations that allow for programmers to convert their code from

[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/

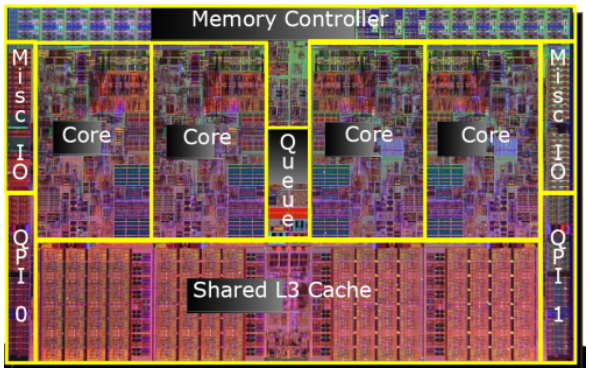

Fig. 1: Quad-core Intel Core i7 CPU Architecture Diagram

Fig. 2: NVIDIA Turing GPU Streaming Multiprocessor Architecture Diagram

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's 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's 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 built-in highlighting of potential speed-bumps to this project's progression.

B. Source Control

Github

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

Fig. 3: The Proposed Timeline of this Project

C. Hardware and Software Setup

1) Visual Studio Code:

- Cross-platform
- Automatable
- Extensions

2) Hardware: For local testing

- Intel® Core™ i7-6700K CPU, clock speed of 4GHz. It is a quad-core CPU with hyperthreading enabled for an apparent core count of 8.
- NVIDIA GeForce GTX 1070 graphics card. Pascal architecture that supports the CUDA Compute functionality, 1.5Ghz base core clock, 8GB of GDDR5 memory, and 15 streaming multi-processors.
- 16GB of DDR4 RAM

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 with having an Intel Core i7-6700K. 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.

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 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 [20] to allow for bi-directional inter-process communication, regardless of how those processes are mapped to the underlying hardware.

Whilst provided in an operation state, this provision is predicated on the correct resources and dependencies being installed and linked up on the executing system. Installation of Intel's MPI libraries

2) Execution:

C. gprof

D. Learning CUDA

IV. Design

- Mathematical model of ideal circumstances
- Code analysis
-

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

A. Mathematical Model

B. Test Harness

V. Implementation

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

VI. Testing

- Test Plan
- Local Testing

VII. Conclusion

A. Limitations

B. Improvements

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