Translating SNAP Algorithm from OpenMP to 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 actually executing the larger program. In this interim report, it is sought to examine the effects of altering the codebase of one such miniapp called "SNAP" from utilising only traditional processing cores to utilising a combination of computational processing cores and graphical processing cores and capabilities (mainly the latter), ultimately examining if a boost in performance can be yielded and justifying if this change can be reflected in the larger-scale software SNAP represents.**

To this end, a brief introduction to the field of High-Performance Computing (HPC) is offered, along with background of the new field of miniapps. 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

[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/

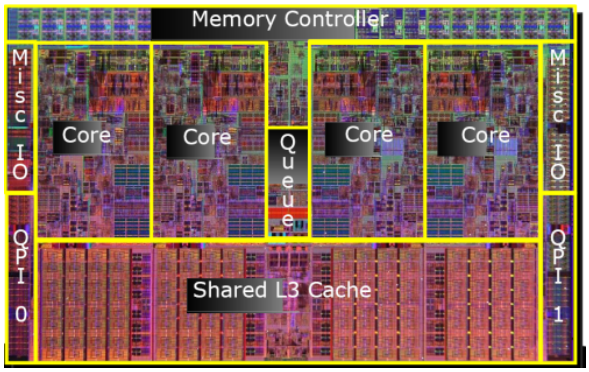


Fig. 1: Quad-core Intel Core i7 CPU Architecture Diagram

Fig. 2: NVIDIA Turing GPU Streaming Multiprocessor Architecture Diagram

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'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 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.

II. 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.

III. Project Review

In this section, we review the current progress made on the project so far and reflect upon how the project has progressed to this point, what has gone well and potential areas for improvement. A section follows discussing the plan for how the rest of the project will progress, outlining key stages, timeline and built-in contingency periods in order to complete the work.

A. Current Progress

The project has progressed well into the planning and design stage. Unfortunately, it is to be said that this is behind the schedule outlined in the project proposal and presentation given in May – a key item highlighted in the Reflection section that follows. Several key areas of consideration and development are underway, categorised as follows:

1) Setting up of a local development and testing environment
2) Developing a practical understanding of CUDA
3) Consolidation of the important facets of the algorithm into psuedocode
4) Development of a mathematical model for how the GPU-based version will operate under ideal conditions

1) Local Development and Testing Environment: In order to maximise the value and amount of time that can be spent developing the project, it is crucial to develop it on a local, non-networked machine to avoid queueing for resources and latency with file transfers and the like.

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

A personal computer was allocated for the task because of its preferential components that can lend great speed and analytical power to the task. Its specifications are as follows:

- 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

This will enable a good baseline assessment of how the SNAP program works on the system for a set of given inputs that can then be used for comparison to the GPU-based approach used on the same set of inputs.

For compilation of the code and to imitate a Linux-based computer cluster, Ubuntu will be used as the base operating system due to its wide-ranging support and package compatibility, in addition to having the following packages installed:

- Gnu C and C++ libraries – for compilation of C/C++ programs, as well as associated tooling such as `backtrace`.
- `mpicc` to compile the standard C-port of SNAP
- `OpenMP` to compile the standard C-port of SNAP
- NVIDIA drivers and CUDA libraries
- All supporting libraries that everything else depends upon

It was important for the program to be run as fast and "as close to the metal" as possible to get accurate results from various executions because virtual machines or Docker containers could add unseen overheads and not give a true indication of speed or performance of the programs.

After the program has shown promising results (if any) on the local environment, it will be submitted to a computing cluster at the university and see if the solution scales with larger inputs. It's possible that the program won't show much improvement in the local environment as it could only marginally improve the performance, whereas it could scale up and save more time per calculation over a much larger input – the type of GPU the cluster runs and the relative amount of streaming multiprocessors in the GPUs versus the number of CPU cores/threads available has the potential to be a significant contributing factor to the results.

2) CUDA Learning: A core part of the progress made has been in learning and experimenting with the CUDA framework which is needed to execute the program on the GPU. It was covered at an introductory level as part of the CS402 High-Performance Computing module. From this, I independently explored the documentation and framework further with the aid of NVIDIA's Developer Blog[6], as well as harnessing additional resources online to gain a clear understanding of the technology.

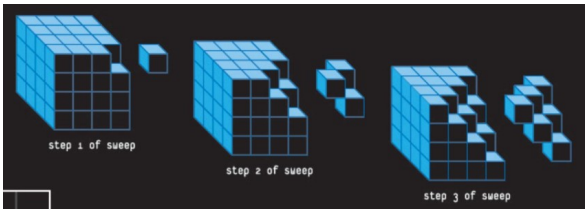


Fig. 3: Slices of a sweep of 3D grid

With this knowledge in mind, as well as details about the architecture and layout of the graphics cards utilised in both the local and cluster environments the program is to be executed in, certain design implementations, such as data layout, quantity of data to transfer, and amount of blocks can be processed in parallel, can be confidently estimated and will guide the direction and effectiveness of the implementation.

3) 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 3, 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 being combined, forms the basis of the next sweep.

[6]https://devblogs.nvidia.com/even-easier-introduction-cuda/

[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

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:

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

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 [20], 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. Reflection of Current Efforts

Three areas come into focus when reviewing the current state of the project:

1) Presentation: Feedback from the presentation was positive, reinforcing and supporting the background research done earlier in the process. This gives me confidence going forward that the project will have a meaningful and relevant outcome to the field. The main issue highlighted with the project was the timeline I had originally set to do each stage of the project. When viewed in isolation, the timeline was generous and appropriate, but factoring in design decisions, examinations, and other deadlines, it became apparent that not enough contingency was built into the process for unseen eventualities and this has unfortunately resulted in the project being behind its original timeline.

2) Examinations: One key reason was the error in not factoring in appropriate time for revision and examinations, which took up the bulk of May and early June, in which the project's development faltered. This is a minor setback and a revised Gantt chart for future development is presented later in the report. Development didn't stop however, as the development environment was still established and, as part of the CS402 High-Performance Computing module, there was substantial headway made with regards to learning CUDA, architecting, and analysing parallelised programs. This knowledge stands me in good stead to progress the work back up to where it should be and at a more fundamentally sound level.

3) Other: Finally, an area that might need to be reviewed, even if just at a cursory level, is the Related Work section. The field of HPC is a fast moving one so, combined with NVIDIA's announcement of their "Super" consumer graphics cards[9] and other technologies, it would

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

[9]https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-super/

be beneficial to review the literature again to ensure that this project is still at the cutting-edge – if new papers and results have come to light, it can only help to bolster this project and the speed and effectiveness with which it can be produced.

IV. Plan for Further Work

Below is an itemised list, in order of what needs to be completed first, of the rest of the work needed to complete the project. The work spans all areas of conceptual design, development, testing, and reporting, due to the time impediments discussed in the reflection prior.

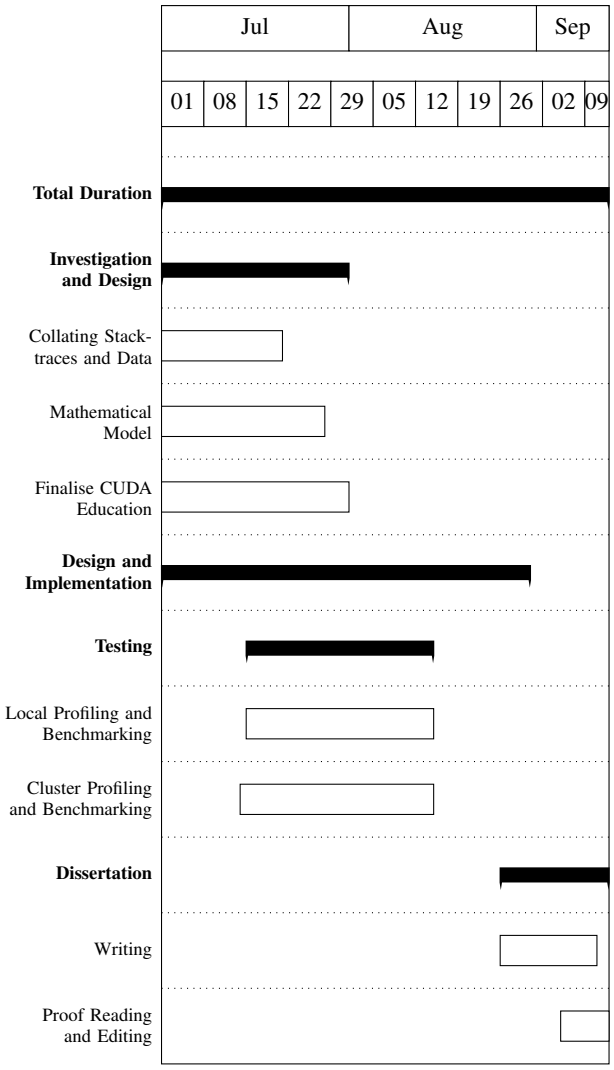
1) Exploring and confirming the mathematical model of the system to ensure that it's representative of what is happening in the system, as well as ensuring it's at least theoretically better than the current CPU-bound implementation.
2) Using the mathematical model as a guide for the ideal scenario, start to develop the CUDA-based alteration in earnest. Care being taken at this stage to write an appropriate kernel function and manage data transfer to be as minimal as possible.
3) Continually test the approach in a local testing environment that allows for parallelised C code to run on a CUDA-enabled device – ideally the bare-metal Linux environment setup on my personal machine but a VM with more than one CPU core allocated and GPU pass-through would suffice.
4) Once development has reached its final stages, collate a range of inputs for the program (provided in the original SNAP repository) and execute them both with the old SNAP code and the new GPU-accelerated program.
5) Compare and contrast the results of step 4, making any necessary design tweaks or fixes to the implementation as required.
6) Fully document the findings in the final dissertation, alongside the methodology, details of the design, and implementation nuances or difficulties faced.

A. Project Management

The code for the project will be managed by a public code repository hosted on GitHub at the following address: https://github.com/alshort/SNAP. It will be hosted under the same licence at the original and follow the usual best practices with regards to version control of atomic changes and experimental items done on a new branch and merged down to master if successful. Version control will keep a backup of different stages of the code should reversion be required. It will also be open-source allowing others to contribute to it in the future.

Finally, the project will be managed at a higher-level by producing work in-line with the revised timeline (as seen below), in addition to frequent communication with the project supervisor and any additional people as required to help provide solutions to impediments, feedback on current progress and the overarching approach, and for further advice should it be needed at any stage of the project.

The Gantt chart of the rest of the timeline of the project is as follows:



References

[1] MET Office. Unified Model, 2018.

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

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

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

[5] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[6] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[7] NVIDIA. Cuda toolkit documentation, 2018.

[8] David 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*, pages 836–838. IEEE, 2008.

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

[10] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[11] Jeffrey A Favorite. A brief user's guide for PARTISN. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2011.

[12] AC Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. *The Cray User Group*, 2013, 2013.

[13] Nathan DeBardeleben, Sean Blanchard, Laura Monroe, Phil Romero, Daryl Grunau, Craig Idler, and Cornell Wright. Gpu behavior on a large hpc cluster. In *European Conference on Parallel Processing*, pages 680–689. Springer, 2013.

[14] Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, and Sharon C Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 2015.

[15] Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael PH Stumpf, and Chris Barnes. Gpu accelerated biochemical network simulation. *Bioinformatics*, 27(6):874–876, 2011.

[16] Christopher Baker, Gregory Davidson, Thomas M Evans, Steven Hamilton, Joshua Jarrell, and Wayne Joubert. High performance radiation transport simulations: preparing for titan. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 47. IEEE Computer Society Press, 2012.

[17] Hayk Shoukourian, Torsten Wilde, Axel Auweter, and Arndt Bode. Predicting the energy and power consumption of strong and weak scaling hpc applications. *Supercomputing frontiers and innovations*, 1(2):20–41, 2014.

[18] Matt Martineau and Simon 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*, pages 185–200. Springer, 2017.

[19] OE Bronson Messer, E DAzevedo, J Hill, Wayne Joubert, S Laosooksathit, and A Tharrington. Developing miniapps on modern platforms using multiple programming models. In *2015 IEEE International Conference on Cluster Computing*, pages 753–759. IEEE, 2015.

[20] Pci express - wikipedia.