Stencil Code Optimization for GPUs Through Machine Learning

Adam Barker
University of Colorado at Colorado Springs
abarker2@uccs.edu

Abstract—**The microprocessor field today has begun to reach its limits as power and thermal constraints have been met and no longer can much leverage of increasing the processor's clock speed be achieved. Thus, much of the scientific and engineering community has shifted to using many-core architectures, such as GPUs, in order to do parallel computations. This paper focuses on the use of genetic algorithms to guide the optimization of stencil codes on NVIDIA's Compute Unified Device Architecture (CUDA) based GPUs and GPGPUs. In particular, we have implemented two separate stencil kernels (Jacobi 7 point and 27 point) in CUDA with each implementation parameterized for several optimiation parameters (thread blocking and loop unrolling factors). We then used a genetic algorithm to find optimal configurations for each kernel. This genetic algorithm is one part of our proposed solution of using an optimization framework incorporating the genetic algorithm to auto-tune automatically optimized stencil codes. Our results show that using a genetic algorithm to auto-tune stencil code optimizations is a valid approach of generating near-optimal configurations in a much more timely fashion than an exhaustive search.**

I. Introduction

As microprocessors reach the power wall, benefits of increasing the clock frequency are no longer achieveable as the cost to system stability and cooling is too much to warrant the increase in performance [1]. This has shifted the focus of the parallel community to many-core architectures, such as those found in Graphical Processing Units (GPUs), as they are comprised of a few hundred or thousand simple cores that are capable of performing highly-parallel computations with much more throughput than a typical multi-core system. However, developing parallel algorithms for GPUs can be no simple task for developers as developers must have a firm understanding of the underlying architecture and hardware properties in order to correctly write programs that correctly take advantage of these properties. Thus, there is a desire to develop a method to automatically apply optimizations to GPU programs in order to avoid the necessity of understanding the complexities of the hardware and architecture of the system.

Recently, in order to meet this desire, researchers have devloped several methods in order to automatically tune or automatically generate optimized codes for both GPUs and multi-core systems. However, as more optimizations are discovered, the search space the auto-tuner must search through grows to an amount where auto-tuning is no longer viable as the number of possible combinations of parameters becomes too large to effectively search through. This then sets the perfect stage for a machine learning application to predict the optimal code instead as it does not have to go through the entire search space, but rather make predictions based on previous results.

This work presents a method to use genetic algorithms in order to discover optimized configurations of parameterized CUDA stencil (nearest-neighbor) codes – a class of algorithms that typically work in structured grids to perform computations, such as finite-difference methods for solving parital differential equations, on a node within the grid by doing computations on the neighbors around the given node. Our work focuses on a simple 3D heat equation using two different stencil codes as the training set for a genetic algorithm to search through a search space of several thousand combinations of possible optimization parameters. Although stencil codes are important as scientific computations, they also provide a unique opportunity for hardware benchmarking as they are computationally simple and require a large use of memory, allowing for benchmarking of instruction-level and data-level parallelism [3]. These codes greatly benefit being run on GPUs as the parallel forms of these codes contain a great deal of instruction level parallelism which translates well to SIMD architectures, which are present on GPUs.

This research is the development of the optimal configuration generator portion of the framework detailed in Figure 1. The auto-optimization framework will be used to optimize existing stencil codes using machine learning in order to predict optimal tuning parameters that will be given to the optimizer which will apply these optimizations to the given stencil code and then output the optimized version of the given code. This is done so that developers can easily write unoptimized code for use in their programs and then run this auto-tuning framework on their code in order to use optimized code that correctly fits within their existing program.

In order to train the machine learning portion of the configuration generator, we implemented two stencil kernels to be used across three separate GPUs. The stencil codes we implemented were a 7 point and 27 point Jacobi iterative stencil codes and then parameterized the relevant optimizations that the genetic algorithms would find configurations for so that the fitness test for the genetic algorithm could change these parameters easily before compiling and running.

The optimization parameters considered for the generation of the search space that we used were the number of threads to use in the computation, and the distance to unroll the inner loop in our code. This inner-loop arises from our use

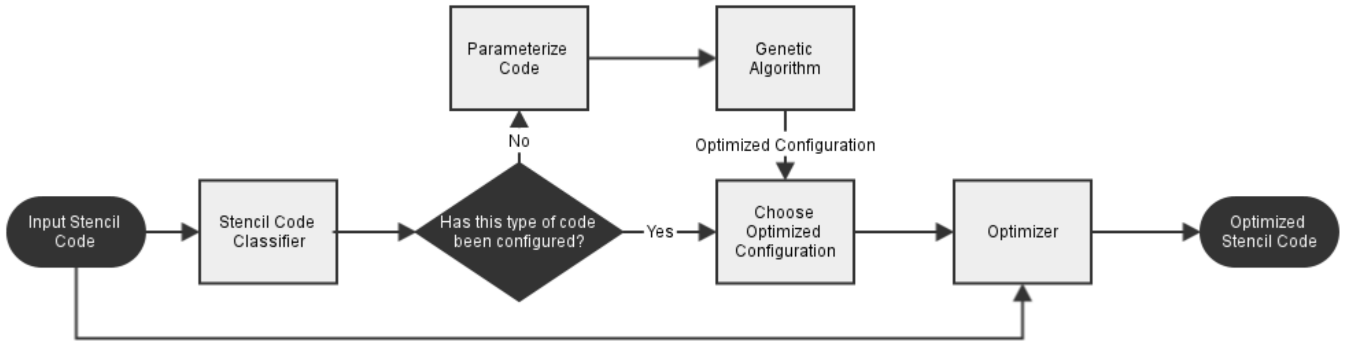

Fig. 1. Overview of auto-optimization framework

of 2.5D blocking, a thread blocking optimization that allows for threads to only be launched in a single plane of the 3D data, and then stream through the remaining axis as the computation goes on. This search space consists of 60 different thread configurations and 192 different loop unrolling configurations, giving us a search space of 11,520 possible combinations. Although the size of this search space is relatively small for most machine learning applications, one must consider the time it takes to compile the code as on our system, typical complilation time is 3 seconds, meaning that an exhaustive search through the search space would take more than 9 hours, whereas our use of a genetic algorithm took on average 8 minutes to find an optimized configuration.

Our contribution is a genetic algorithm that is capable of tuning optimizations on parameterized stencil codes. This genetic algorithm can effectively tune these codes to find near-optimal configurations for the applied optimizations in a very short amount of time, making it an effective method to use for auto-tuning stencil code optimizations.

The rest of the paper is organized into four sections: related work, tuning framework, experimental results, and conclusions and future work. In related work, other research that has been done in the field is presented and summarized along with how it is utilized in this research. The tuning framework section goes into more detail of stencil codes, optimizations, and the genetic algorithm that we used. Experimental results includes the experimental setup and the results we obtained from running our implementation on three different systems as well as a discussion of these results. Conclusions and future work summarizes this research and presents the outlook of incorporating it into future work.

II. Related Work

There exists significant research to automatically tune optimized stencil codes in order to find the best configuration of parameters for such optimizations [1], [3], [8], [11]. Datta et all have demonstrated the usefulness of optimizations with auto-tuning techniques as a means to effectively optimize stencil codes on both CPUs and GPUs [3]. Their work provides an effective base for the challenges of optimizing and auto-tuning stencil codes. Gana et all cite this work as their basis for using machine learning to optimize CPU stencil codes. In their research, they used a genetic algorithm in combination with the KCCA algorithm to perform quick searches through the parameter space of 4×10^7 different combinations. They managed to effectively auto-tune stencil codes on CPUs in two hours using their method [5]. Zhang and Mueller also researched auto-tuning and auto-generation of optimized stencil codes specifically for GPUs and GPU clusters which provides a more specific list of optimizations that are specifically used for GPU stencil code optimizations that were used in this research. In particular, their descriptions of 7-point and 27-point stencils, along with shared memory and register allocation for optimization were used throughout our research. [11].

Many optimizations have been developed over the years for stencil codes [2], [6], [7], [9], [10]. Nguyen et all provided a state-of-the art stencil code optimization that uses a combination of 2.5D thread blocking combined with 1d temporal blocking to create what they have called 3.5D blocking which provides throughput increases on GPUs of about two times what prior research had claimed [10]. In our research, we used their excellent description of 2.5D blocking as one of our optimizations for the genetic algorithm to automatically tune. Nguyen et all's research can also be parameterized by changing the amount of temporal blocking to perform, thus allowing a search space to be created for this optimization which was incorporated into this research.

III. Optimization Framework

A. Stencil Codes

Stencil codes are primarily used to solve partial differential equations in order to perform simulations such as heat flow or electromagnetic field propogation [3]. Most methods for solving these partial differential equations use iterative sweeps through spatial data, performing nearest-neighbor computations which are called stencils. Each node in the computation is weighted based on distance from the central node, which allows for the solving partial differential equations by switching these weights for the coefficients used in the solver. Using this structure, methods are created for different types of partial differential equation solvers such as Jacobi iterative methods, which are the stencil codes we used in this research.

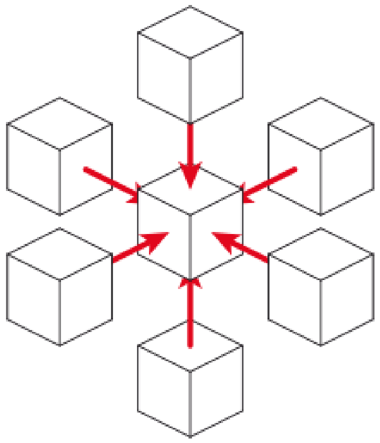

Fig. 2. A 6-point Von-Neuman stencil (credit: wikipedia.org)

As the data sizes used for stencil computations typically range outside the size of available cache memory, there is a large emphasis on data reuse and data-level parallelism in order to fully optimize stencil codes. This can cause portability issues as memory speeds and sizes can differ widely system to system, causing the need to use different parameters for optimizations on different architectures. This then produces a demand for a method to automatically tune stencil code optimizations on each architecture in order to enhance portability of the codes.

Auto-tuning of stencil kernels has become a fairly large area of study in order to work around the necessity of knowledge of the low-level specifications of the architecture in order to optimize the kernel. However, these auto-tuners may have to look in a parameter space that is upwards of 40 million combinations that may take months to fully check every single one for optimal performance [3]. This then creates a demand for a faster optimization process that is still automated in order to create a process that is viable for industry use. Thus, machine learning may be a good option for automatic optimization as it can use reinforcement learning paired with statistical machine learning and genetic algorithms in order to explore the parameter space much faster. Using machine learning may also overcome another downfall of auto-tuning in that each auto-tuner is generally programmed for one architecture, whereas a learner can learn architectures as well and correctly optimize for them.

B. Optimizations

In this research, we applied two types of optimizations to our stencil codes to be used in the tuning phase. The first optimization we used was 2.5D blocking. 2.5D blocking is an optimization for thread blocking of 3D stencil codes that only blocks in the x and y axes of the structured grid. Each thread then streams through the remaining z-axis, allowing for data-reuse of data already fetched by the thread earlier to fulfill data requirements. This optimization reduces the amount of global store and load instructions as threads can keep some data in the registers for quick access for several computations instead of fetching data from global memory each time a node must be calculated. The second optimization used is loop unrolling. Due to the nature of 2.5D blocking in that it must stream through the z-axis via a loop, this loop can be unrolled in order to provide more data-level parallelism and keep threads from becoming idle. These optimizations must be tuned in order to be fully optimized. 2.5D blocking takes two parameters: an x-axis blocking dimension and a y-axis blocking dimension. For a 256^3 grid, there are 60 different configurations of 2.5D blocking. For loop unrolling, the maximum unroll length allowed by the compiler is 192 iterations. By combining these two search spaces, the genetic algorithm used for tuning these optimizations searches through a search space containing 11,520 different configurations.

C. Genetic Algorithms

Genetic algorithms are a set of algorithms that mimic the natural selection process in order to find solutions to problems. Genetic algorithms do this by generating an initial population that generally consists of randomly generated individuals that contain randomly generated values for each parameter that will be searched. Each individual in the population then undergoes a selection process by which the fitness of their parameters that they contain is evaluated. The most fit individuals are then selected to be mutated and mated with each other in order to generate the next generation of individuals. This then continues until the population either converges to a singular value or the number of set generations is reached.

In this research, we used an initial population of ten individuals each with a chromosome (parameter set) containing three parameters – thread blocking for the x and y axes and loop unrolling factor. This population then underwent ten generations in order to get the individuals to converge on one value. The best performing individual was saved and then returned at the end of the generation process as the best configuration for the given optimizations. All of this was done using the Distributed Evolutionary Algorithms in Python (DEAP) project [4]. It allowed for use of built-in algorithms for the mating, mutating, and selection processes.

IV. Experimental Results

A. Goal

The goal of this experiment is to determine if genetic algorithms are a viable approach to tuning stencil code optimizations faster than other methods of tuning. The genetic algorithm used must be able to produce an optimal or near-optimal configuration for the optimized stencil code in a reasonable amount of time in contrast to the time it takes for an exhaustive search method to find the optimal configuration.

B. Setup

The setup we used to perform the experiments on consisted of three GPUs: one GPGPU (Tesla C2050) and two standard

GPUs (GTX 480, 680). Figure 3 details the theoretical peak FLoating-point OPeration (FLOP) rate determined by the number of cores (α) multiplied by the clock rate of each core (δ) multiplied by the number of FLOPs that can be performed each clock cycle (γ).

$$\alpha \times \delta \times \gamma = GFLOPS/sec$$

Fig. 3. Theoretical peak FLOP rate equation.

GPU	Architecture	Peak FLOP rate
GTX 480	Fermi	1344 GFLOP/s
Tesla C2050	Fermi	1030 GFLOP/s
GTX 680	Kepler	3250 GFLOP/s

Fig. 4. Peak GFLOP rate of GPUs (single precision)

The genetic algorithm was run on two stencil kernels: a 27 point Jacobi stencil and a 7 point Jacobi stencil. This genetic algorithm was used on each of the GPUs and was trained on the GTX 480 using the 7 point stencil. After the initial training, no values of the genetic algorithm were changed in order to generate the final results. The genetic algorithm used a three-gene chromosome to find configurations. The first two genes were for thread blocking dimensions along the x and y axes each being a power of two and their combined product could not exceed 2^{10} (60 combinations for 256^3 grid size). The third gene was for loop unrolling which was an integer from 1-192 for unroll length. The combined search space consisted of 11,520 different combinations the algorithm could possibly generate. This genetic algorithm was then run to create ten generations based on an initial population of ten individuals in order to find a configuration for each optimized stencil code that was close to the optimal value that was found through an exhaustive search of the search space.

$$a^{n+1}_{i,j,k} = \alpha(a^n_{i,j,k} + a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1})$$

Fig. 5. 7-point Jacobi stencil equation.

$$a^{n+1}_{i,j,l} = \alpha(a^n_{i,j,k}) + \beta(a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1}) + \\ \gamma(a^n_{i\pm1,j\pm1,k} + a^n_{i,j\pm1,k\pm1} + a^n_{i\pm1,j,k\pm1}) + \epsilon(a^n_{i\pm1,j\pm1,k\pm1})$$

Fig. 6. 27-point Jacobi stencil equation.

The equations in figures 5 and 6 detail a typical 7-point and 27-point Jacobi stencil where a is the input grid, n is the iteration, and $\alpha, \beta, \gamma, \epsilon$ are coefficients multiplied upon the neighborhood sum. The ±1 symbols are used to save space in writing out each $i+1$ and $i-1$ for each i, j, k within the array of nodes.

C. Results

The graphs in Figure 7 are of the average performance in GFLOPS/sec of the population per generation. The red line is of the performance of the 7 point stencil code and the blue line is of the 27 point stencil code. The two dashed lines in each graph show the optimal configuration performance for each stencil code. The optimal configuration was found by performing an exhaustive search through the parameter search space.

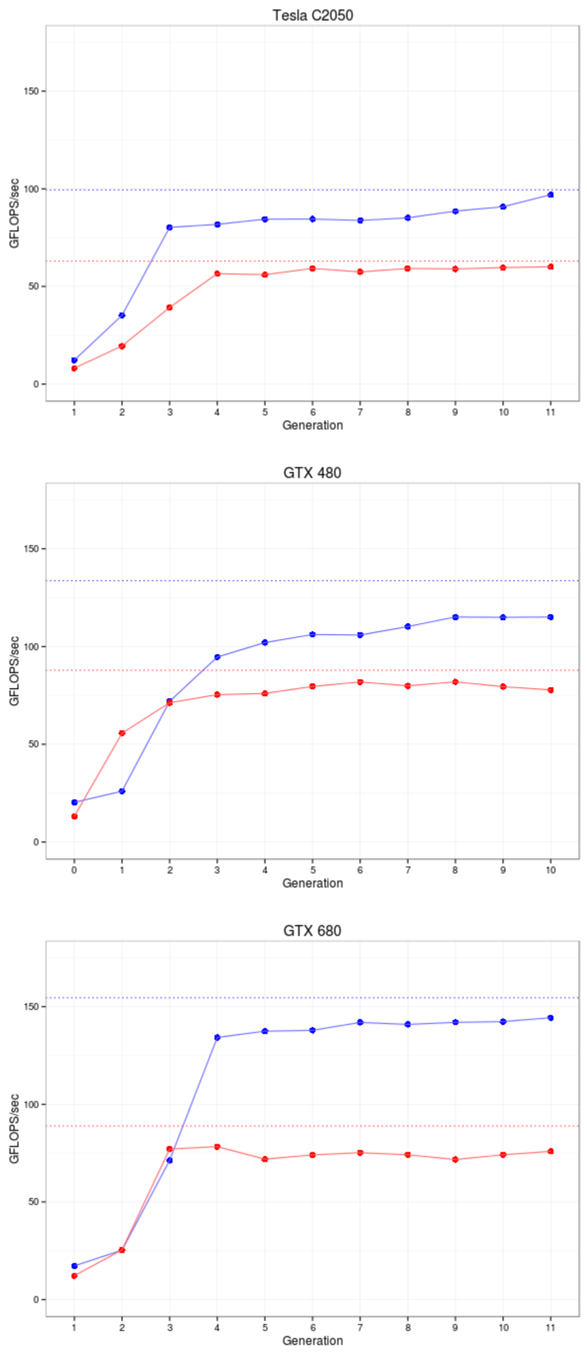

Fig. 7. Genetic algorithm average fitness of each generation for the three GPUs on both stencil kernels. The solid lines are for the average population fitness by generation for the 27-point stencil (blue) and 7-point stencil (red). The dashed lines show the optimal configuration throughput rate.

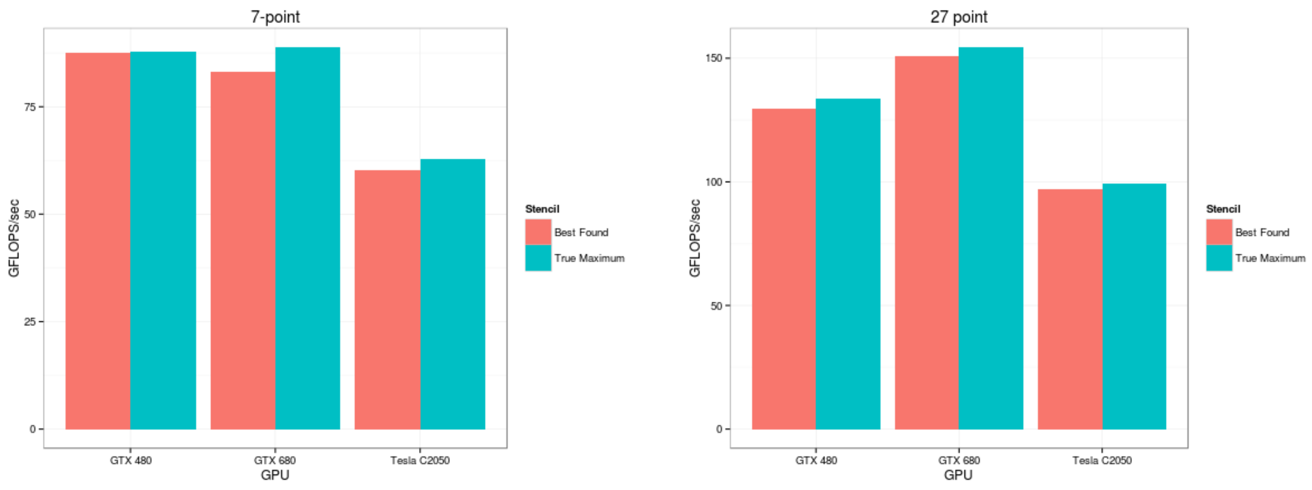

Fig. 8. Best configurations found by genetic algorithm vs the absolute best configuration found by exhaustive search for each stencil and GPU.

These results show the effectiveness of a genetic algorithm approach to auto-tuning stencil code optimizations as it generally only took 3–4 generations for each stencil code to be near-optimal. It should also be noted that these results only show the average performance of the entire population per generation, not the best candidates. The best candidates shown in Figure 8 of the population were typically within 3% of the optimal performance found for each stencil kernel by the 10^{th} population. The initial population for the genetic algorithm consisted of only ten members. Due to the small search space size, this small number of members was still able to quickly converge to a near-optimal configuration for each kernel. The small search size also allowed for us to check our results through exhaustive search as doing so took about 4–5 hours per kernel for each GPU. This speed is in contrast to the average eight minute execution time for the genetic algorithm to generate all ten generations and find a near-optimal configuration. These speeds differ in terms of which CPU is used to compile each code, but the large gap in performance still persists for each CPU, regardless of its speed.

However, these results show that each kernel could only reach up to a maximum of 100 GFLOPS/sec for the 27 point stencil on the Tesla C2050, which is far lower than the 450 GFLOPS/sec produced by Bergstra et all [2] on the same model of GPU. This is due to the optimizations that were used in our stencil codes as they are the main bottleneck of performance for the stencil code. Our optimizations still contain thread divergence in the code, and is thus less optimized compared to Bergstra et all's kernel which contains no thread divergence. For future work on this research, more optimizations will be considered so that the results will be closer to current stencil code performance.

V. Conclusions and Future Work

This work demonstrates the effectiveness of using a genetic algorithm in order to find near-optimal configurations for stencil code optimizations across multiple GPUs with differing architectures. This result allows for enhanced portability of stencil code optimizations to differing architectures in a timely fashion as the tuning phase was demonstrated to be much faster than exhaustive search alternatives as the genetic algorithm took, on average, eight minutes to generate all ten generations of the population in contrast to the 4–5 hour run time of the exhaustive search.

In the future, we would incorporate more parameters for use in the chromosome for the genetic algorithm in order to generate a search space worthy of using a machine learning technique to traverse it instead of exhaustive search being a viable method to use. We will also develop more parts to the auto-optimization framework from figure 1 such as the optimizer and stencil code classifier. This is in the hopes that a functional framework can be created such that it may be used to optimize existing stencil codes that are in use today and be continually optimized as more stencil code optimizations are found.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. 2012.

[3] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[5] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar09)*, 2009.

[6] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[7] Julien Jaeger and Denis Barthou. Automatic efficient data layout for multithreaded stencil codes on cpu sand gpus. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[8] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, page 256265. ACM, 2009.

[9] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM.

Acknowledgements

This research is supported by NSF grant 1359275.