GPU-Acceleration of Radar Propagation Models

Project Paper

Sandy Van Eyk
Student
The University of Adelaide

Thomas Woolford
Industry Supervisor
Swordfish Computing

Olaf Maennel
Academic Supervisor
The University of Adelaide

ABSTRACT

Modelling the propagation of electromagnetic (EM) waves over terrain is computationally expensive. This limits the quality of models and accessibility of high-quality models. GPU-acceleration has the potential to make these computations orders of magnitude faster. This allows for the design of better EM systems such as radar and telecommunications networks.

1 INTRODUCTION

This paper aims to simplify the creation of and improve future GPU-accelerated radar propagation simulations. To do so it fills a gap in the current literature where GPU-accelerated EM wave propagation simulations are explored concerning a specific method, GPU programming interface, and application. They do not discuss general applicability of the improvements or the process. This limits the applicability of the research, reducing its impact on simplifying the implementation and advancement of the used, and other methods, regardless of the application or API used. Thus this paper focuses on the process of preparing for and GPU-acceleration with discussion of decisions, trade offs, issues, and challenges.

The project's scope is limited to the FDTD[1] method in 2 dimensions due to time constraints. FDTD is chosen because it shows promise in the desired modelling while being simple. This allows for a focus on general GPU implementation considerations as opposed to optimising and understanding a specific algorithm. Further, it allows for the collection of higher quality results in the same timeframe.

[1] Finite-Difference Time-Domain, a method for modeling EM fields over time.

This paper has six primary sections, background, literature review, methodology, performance, discussion, and appendix. The background section conveys further motivation behind the project and required knowledge for reading this paper. The literature review displays the gap through a review of relevant papers and their limitations. The methodology describes the process used to perform this research and arrive at the results. The performance section summarises performance figures obtained. The discussion section explores how the results are generally applicable and their implications for future projects implementing GPU-acceleration of radar propagation. The Appendix generally explores the specifics of this report only relevant to the FDTD method. It reads most clearly when jumped to upon reference throughout the paper.

2 BACKGROUND

The modelling of radar waves propagating over terrain allows for prediction of how they act in real-world systems. With this prediction, efficient radar networks with higher strength and more complete coverage can be designed and built. This directly benefits defence, telecommunications, networking, and any other sectors that operate at similar radio frequencies. The modelling methods are computationally expensive.

2.1 Radar and FDTD

RADAR (Radio Detection And Ranging) transmits EM waves typically in the range of 30 MHz to 8 GHz depending on the application. A receiver then receives waves that reflect off objects and by timing the duration from sent to received, the distance to objects can be determined. Doppler shift, a change in reflected frequency due to object

movement, can be measured by the receiver to determine the velocity of objects [1]. This principle can be used to separate stationary objects from non-stationary ones.

Modelling radar propagation is achieved through modelling the EM waves' interactions with physical objects. In the case of FDTD, this is done by approximating Maxwell's equations on a grid. At each cell, the electric and magnetic field components are staggered by half a cell. The electric and magnetic fields are then evaluated half a time step apart [2]. The electric field and magnetic field cells are each updated according to their previous value and the surrounding values from the opposite field. Absorbing boundary conditions, ACBs, are used to make the edges of the simulation absorb incoming waves and thus act as if the simulation space continued into free space.

FDTD and other methods are explored in more depth in the book, *Computational Electromagnetics for RF and Microwave Engineering* by David B. Davidson.

2.2 Computer architecture

At a high level, high level code is compiled to instructions (machine code). These instructions are then run as a process on processing units in a computer. Modern processors have many processing units, cores, each capable of independently processing these instructions. Processes also consist of threads. A thread is a . Multithreading is the use of multiple threads concurrently.

Processors also have caches to temporarily store information relevant to the processor. This is done to reduce the penalty for memory access. Memory access usually requires significantly more time than typical operations. This delay is significantly reduced by having relevant memory in a faster cache. For cost efficiency, caches are typically layered with smaller faster caches, closer to the processor (lower level). Upon memory access the cache is checked for the required info and it has it then there is a cache hit. If it does not have the info there is a cache miss and the higher level of cache or memory is checked. with a miss on a lower level accessing a higher level cache until . Lower level caches have

GPU-acceleration is the use of a GPU alongside a CPU to offload work from the CPU to the GPU, allowing for faster execution of code. The functions to run on the GPU are referred to as kernels. Offloading work from the CPU to a GPU can improve performance primarily due to the vast difference in core count. For example, the current highest core count workstation GPU has 18,176 cores vs the equivalent CPU product with 96 cores[2] [3-4]. This makes GPUs significantly faster than CPUs at processing highly parallelizable workloads. Workloads can be considered parallelizable where tasks or iterations produce correct results when running concurrently. GPUs are not used for all computations as they are optimised for parallelism. This makes general computer operations that are sequential and require context switching more suited to the CPU.

2.3 GPU architecture and related optimisations

Where differing, CUDA[3] terminology is used throughout this section. The same concepts apply to other interfaces under different terms with generally the same architecture.

Broadly, GPUs consist of many streaming multiprocessors (SMs). Each of the SMs has many cores and a shared memory for storing and sharing data between cores. These cores are where the threads will run. Multiple levels of cache are also located throughout the GPU.

When executing a kernel, many thousands of threads are created. These threads are individually identifiable allowing for "assigning" threads to data. The threads are split into groups called warps. Each warp must execute the same instruction at each cycle. The threads are also split into blocks. A block should contain multiple warps to be performant. The SMs are assigned blocks and can have multiple blocks in progress at once. However, the blocks can have only one SM. Threads inside a block can be synchronised and interact via shared memory. Blocks cannot interact with each other. From this architecture, several interactions and possible optimisations arise.

One such interaction occurs if instructions are conditionally executed in the kernel, i.e. an "if" statement. If threads in a warp take different paths of a branch, the branches are effectively executed serially. Therefore if

---

[2] Nvidia RTX 6000 and AMD Ryzen Threadripper PRO 7995WX respectively.

[3] CUDA is a GPU programming interface for NVIDIA GPUs.

branches are required they should be implemented such that all threads in a warp take the same path.

A side effect of instructions executing on a per-warp level is that memory access does the same. To be performant, threads in a warp must access adjacent memory locations. By doing so when memory is accessed as a block all threads can fetch their related data. This is referred to as memory coalescing. In the worst case without it, each thread requires its own memory access, effectively running all threads in the warp in serial.

An optimisation utilised by GPU hardware is by storing many warps on each SM at once there is a minimal cost of switching context. When a warp is stalling such as when accessing memory on cache misses, other warps can be swapped to and executed. This allows work to be done while waiting for memory, effectively hiding the memory latency. To support this it is beneficial to "over-subscribe" the GPU with greater warps and blocks than there is hardware to process at once.

Threads can communicate within a block using the shared memory. Compared to global memory it has significantly higher bandwidth and lower latency. This also allows it to be used as a programmable cache where data used multiple times within a block can be stored.

3 LITERATURE REVIEW

Multiple methods for modelling radar propagation show performance improvements with GPU-acceleration [5-9]. The papers where this is done focus on the implementation for their scenario with little to no focus on the general process of GPU-acceleration and the decisions and tradeoffs that come with it.

An example is the 2021 paper by Liu, Yongjun, et al., *Accelerating the Simulation of Finite Difference Time Domain (FDTD) with GPU*. In it, the researchers explain FDTD, CUDA, the implementation, and the performance gain. They took the simulation time from ~12 hours with 2 CPUs (20 cores each) to ~30 minutes with 4 GPUs (5120 cores each). There is no sufficient explanation behind how they chose to parallelise FDTD.

A similar paper published in 2022 is the *Acceleration of FDTD Code Using MATLAB's Parallel Computing Toolbox*, Alec Weiss, et al. The researchers use MATLAB to implement the GPU-accelerated FDTD code. In the paper, they discuss several MATLAB-specific methods for GPU-acceleration. They also provide the full source code and detail how it works. Further, it contains an explanation for optimising FDTD data shared between multiple cells. It discusses the distribution of the FDTD domain across GPU cores but only in one dimension for simplicity. The researchers compare performance in computed cells per second with varying numbers of CPU cores and numbers of GPUs. They do not compare between CPUs and GPUs.

A highly relevant paper is *A CUDA-based GPU engine for gprMax: Open source FDTD electromagnetic simulation software*, Craig Warren, et al. It explores the GPU-acceleration of FDTD for simulating ground penetrating radar. The researchers discuss the design of kernels, related optimisations, and the performance between different CPUs and GPUs using time-based metrics. The discussion on design decisions goes in-depth on alternatives but only looks at it from the bounds of this algorithm. They also discuss what the algorithm is bound by and how this relates to their results.

So far only papers using the FDTD method have been discussed. There are also numerous papers showing GPU-acceleration's viability with other methods. For example, GPU-acceleration improves performance in the MoM and FEM[4] methods as displayed in papers such as those by Badía, José M., et al. [8] and Arduino, Alessandro, et al. [9] respectively. The first achieves a performance improvement of over 140 times compared to sequential. It also goes into some detail about decisions made during parallelisation. The latter speeds up execution by up to 50 times and barely discusses implementation focusing more on testing and the method itself. Both used time-based metrics.

All the explored papers only explore GPU-acceleration concerning the specific method, GPU programming interface, and application without discussing the general

[4] These methods are beyond the scope of this report. More can be learnt about them in the previously mentioned book by David B. Davidson.

applicability of the improvements or the process. This limits the applicability of the research. Another limitation is that any decisions made regarding the implementations are either not discussed or not backed up with any displayed data. Further, all the metrics used to display performance improvements are time-based and thus dependent on hardware.

The papers explored in this section were found through searches on Google Scholar involving keywords such as FDTD, GPU, radar, propagation, EM, RF, radio, accelerated, terrain, and model.

4 METHODOLOGY

Preliminary research is conducted on the FDTD method. This starts with learning how and why it works. By reviewing existing implementations, a single-threaded implementation is created in one and then two dimensions. These initial steps are necessary if not already possessing an understood single-threaded implementation. The specific method used to implement these systems appears to involve short stand-alone stages where the system would be coded with minimal testing and supporting code. In practice, it takes far longer and consists of cycles of implementation, animation implementation, debugging, and then repeat. This is due to the required implementation of supporting features such as plotters, terrain data, sampling, and testers. This more involved process significantly increases time required. Further, if not careful with implementation and module choice, components may have to be redesigned and coded multiple times.

Thus Single-threaded FDTD is implemented in one dimension with first-order ABCs, point sources, and line plotting functionality. It is also implemented in two dimensions with second-order ABCs, Ricker wavelet sources[5], heatmap plotting, and terrain loading functionality. For information on the specific cycles, what is implemented and any challenges, see Appendix 8.3.

To test the implementation with a somewhat realistic simulation real terrain altitude data can be used. The chosen location is not too important so long as it provides reasonable terrain where it is expected for radar to operate. In this report the location[6] is chosen for its bowl-like terrain, local airstrip, locality, and size. The height data is collected from Google Earth by tracing and scaling the altitude of a linear path onto an image. Images of the Google Earth path and resulting image can be found in the Github repository, section 5.1.

Next GPU programming interfaces are researched. This involves the collection of a list of potential GPU programming interfaces for general-purpose computing[7]. This list is narrowed down through the review of necessary and important criteria. Here, OpenMP is chosen for how simple it makes GPU-acceleration. This is at the cost of control and makes later attempts at optimisation more difficult. In section 6.2 the specific criteria used to select OpenMP will be discussed further. The chosen interface is then set up. For OpenMp this may require building gcc with additional features. For this project a precompiled gcc package is installed in a docker container for compilation. The Dockerfiles for this and a mostly complete building of GCC are located in the project repository (see Section 5.1) under 2D\docker_app. There is also a readme with documentation on environment setup and compilation for Windows.

With the chosen interface in mind, plans for system parallelisation can be made. First, multi-threaded operation is considered. This applies to running the algorithm on the CPU or GPU. Then specify the system for GPU-acceleration, using research on GPU architecture to determine optimisations and parameter bounds. In FDTD, aside from the main update loops that can be parallelised, some of the macro tasks can also operate concurrently.

A single-threaded 2D FDTD system does the tasks in Figure 1 sequentially for each step (referred to here as iteration).

[5] A pulsed source capable of introducing broad-spectrum waves.

[6] Wilpena Pound, specifically crossing St Mary Peak

[7] For this project in alphabetical order, this list contains CUDA, Direct Compute, HIP, Kokkos, MATLAB, OpenACC, OpenCL, OpenMP, and SYCL

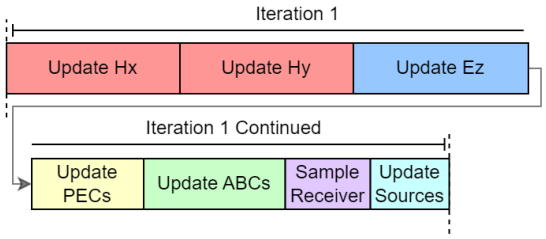

Figure 1: Single-threaded FDTD iteration tasks

By analysing the dependencies between each task, Figure 3 is created to show what tasks can run concurrently with minimal limitations.

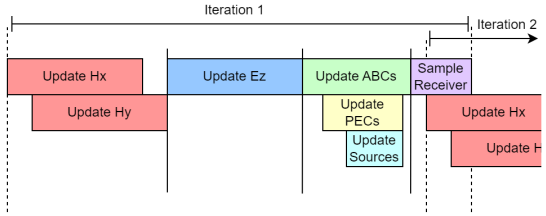

Figure 3: Multi-threaded FDTD iteration tasks

In the figures, dotted lines show the boundaries of each iteration. Solid lines show where tasks in the following stage depend on the previous stage's completion. The offset shows how the tasks wouldn't start at the same time. Note that iterations can now overlap.

Through the research of the intended GPU accelerator,[8] layouts for coalesced memory access and guidelines for simulation parameters are found. The memory layouts, their respective evaluations and reasoning can be found in Appendix 8.1. Between roughly 25 and 200 warps per block, per warp performance is found to be the same in theory. With this design the intended simulation parameters of 30MHz wavelengths with 20 grid points per wavelength, correlate to simulation sizes of 9200 m^2 and 115,200 m^2. Smaller simulation sizes should decrease overall computation time but with increased time per warp. Exceeding it should increase computation time and time per warp. For details on these numbers and their calculation see Appendix 8.2.

Importantly, OpenMP allows for acceleration with GPU or CPU using similar syntax. This is used to create a Multi-threaded (CPU) implementation as an intermediary between single-threaded and GPU-accelerated FDTD. Finally, GPU-acceleration and performance optimisations are integrated according to the design.

With GPU-acceleration performed and an example simulation made, performance metrics can be collected. Here the tester framework is used to test each snapshot of the system, but is not required. The tests should be conducted with minimal background tasks and in the same session to minimise external factors. Execution time is used as a metric to gauge performance. This works well for displaying performance changes due to optimisations. It is less useful for comparing performance between GPU and CPU due to potentially unbalanced systems. Despite this, it is still used as such in this report as reducing computation time is a major purpose of GPU-acceleration.

5 PERFORMANCE

The following performance figures are created through simulation of the Wilpena Pound example. Notably in the testing of GPU results certain features had to be executed on CPU and others disabled due to time restrictions and undefined errors with OpenMP. This does not affect the performance results in favour of the GPU.

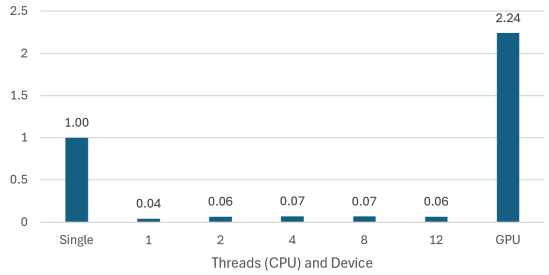

Figure 4: Speedup for devices (195M cells, 100 steps)

These results are quite poor. The multithreaded implementation has significantly worse performance than the single threaded. This is likely due to the minimal time spent to implement it. Therefore the following figures will refer to the single-threaded implementation. The GPU implementation also only manages a 2.24x performance improvement.

5.1 Code and figures

A Github repository contains the code and figures associated with this project. It can be found at:

https://github.com/A1871034/radar-project

[8] NVIDIA RTX 3070

At the time of this report's creation, the project is at release "Report" with the "GPU" tag. The snapshots of the system in its single-threaded, multi-threaded, and GPU-accelerated states are located under branches single-threaded, cpu-multi, and main, respectively.

6 DISCUSSION

While only the FDTD method is explored, the key takeaways, considerations, and implications apply to any radar propagation method. These points are split into two subsections, single-threaded implementation and GPU-accelerating the implementation. The latter of which is more generally applicable. However, the former is still necessary for any implementer without a single-threaded implementation.

6.1 Single-threaded implementation

When researching and implementing the single-threaded FDTD implementation, generally applicable aspects are found.

If creating a simplified implementation first, such as one in a lower dimension, ensure it either remains simple or the work is transferable to the final implementation. It is obvious to keep the simplified implementation simple. However, it is easy to implement more than necessary for learning. Further, shared components between implementations might not share their design. Creating such an implementation can be an inefficient use of time. If the simple version is already understood or can be understood by studying it, implementing it will only provide familiarity.

All radar propagation systems will require the same base framework. This framework should provide features for sampling, testing, loading data into the simulation, and applying sources during runtime. In this report, the framework is realised by exporting simulation data, animating simulation data, transforming images to height data, and a wave source module. Adequate time should be planned for the design and implementation of at least this base framework.

Similarly, if using any external tools or libraries, ensure they are suitable for future requirements. For example, the initial Python Matplotlib animator was significantly too slow for the data sizes that were to be expected later. This required both learning a new plotting library and reimplementing the animator.

The export and then animating system used for verification and debugging is not ideal. It requires two separate systems to interact and on larger simulations takes up a lot of storage. For example, one second of animation at 12fps for a 19500 x 1000 test simulation requires 936MB of storage if storing values as floats. A system that can create frames of the animation in the same code as simulation stepping would be highly beneficial. This would remove the raw data storage requirement and improve the speed of animation by not requiring storage reads.

In testing, it is recommended to export the base metric (eg. time) before calculating further metrics. This avoids any potential complications due to variable types and limitations. During performance testing, it is found that calculated metrics are off by orders of magnitude. This is likely due to floating point precision and typecasting. However, unable to quickly fix the issue, it is deemed better for accuracy, time, and certainty, to export the base metric. The calculations can then be completed externally in software resilient to these errors.

6.2 GPU-accelerating the implementation

The first step for GPU-acceleration is to choose a GPU programming interface. This must be done before designing the implementation. The requirements for the GPU programming interface will be largely determined by factors external to the specific design. A suitable method for choosing the GPU programming interfaces is to filter by selection criteria. As well as any specific criteria this generally includes an interface's GPU vendor support, platform support, and if it is kernel-based or directive-based.[9] For example, OpenMP is chosen for this report following the broad criteria of being generally

[9] Directive-based uses compiler hints (directives) on otherwise serial code to pass execution to acceleration devices such as GPUs. This makes it simple to perform gpu acceleration on existing code. Kernel-based allows writing low level kernels that run on the GPU allowing for more control and higher performance. However, it requires more knowledge of GPU architecture.

applicable. This encompassed other criteria like being platform and GPU vendor agnostic. Further, it is directive-based to show how it can be simple to gain performance with little change to the code. Looking at the documentation before settling on an interface is also beneficial. This ensures that the documentation is clear and the interface provides intuitive and relevant features.

It is also important to leave considerable extra time for setting up the GPU programming interface. If it hasn't been used before this can take significantly longer than expected. Even if the interface is previously used, it is still valuable to leave extra time. Environmental changes such as system updates may introduce unforeseen and difficult errors. In this project that time is not allocated. This resulted in significant delays when discovering that while OpenMP can be compiled with regular GCC, the GPU directives cannot. OpenMP is built with common compilers by default, however, the GPU offloading parameters are not. Significant resources are spent trying to build GCC with GPU offloading enabled. It is later discovered that there are pre-compiled GCC packages with the required features enabled. After installing missing dependencies and determining required extra compiler arguments, the code could now be compiled with OpenMP GPU offloading.

A few of the most important optimisations for GPU performance have already been covered. These optimisations included memory coalescence, warp divergence, warp over-subscription, and caching. While these optimisations are not as relevant to implement for this report, they provide readers who are concerned with performance a great starting point for optimisation.

Another generally applicable optimisation is to reduce the accuracy of the simulation. If it is viable, reducing the accuracy typically provides significant performance boosts. As an example, in this report, double-precision floats (8 Bytes) are used for increased accuracy. Swapping to single precision floats (4 Bytes) results in immediate performance improvements. This halves memory accesses (assuming perfect coalescence) and memory usage. This can result in significant speedups. In maths-bound applications, it can result in speedups up to 64x.[10] Another example is the task parallelism previously explored in Figure 4.2. The tasks, Update ABCs, Update PECs, and Update Sources, are shown to execute concurrently. This limits some functionality of the simulation but allows for performance improvements. The limitation is that ABCs PECs and sources cannot overlap without resulting in undefined execution order and decreased accuracy. The limitation is due to them all operating on the same electric field. Overlapping these components is atypical; the performance improvement is achieved with minimal impact.

7 CONCLUSION

This paper has filled the gap in the literature by exploring and reporting the process of GPU-accelerating the FDTD method in 2 dimensions. It describes the process used to perform this research and summarises key points and outcomes. By filling this gap, future GPU-accelerated radar propagation simulations can be implemented more easily with better results. Future projects have a good starting point of how to choose a framework, possible optimisations, and the challenges to expect along the way. Further, it is relevant for any stage of implementation from no implementation through to a working system that is already GPU-accelerated.

Further research on this topic may involve following a similar method with a different radar propagation algorithm, GPU programming interface, or both. It may also extend the general optimisations presented, to test their efficacy in radar propagation applications.

[10] The speedup of up to 64x is based on the theoretical FLOPs for double-precision floats being 1/64th of the FLOPs for single-precision floats on the RTX 3070. Potential speedup will vary with hardware.

REFERENCES

[1] Jenn, David. “Radar Fundamentals.” Naval Postgraduate School, https://faculty.nps.edu/jenn/seminars/radarfundamentals.pdf. Accessed 28 3 2024.

[2] Davidson, David B. “An Overview of Computational Electromagnetics for RF and Microwave Applications.” Computational Electromagnetics for RF and Microwave Engineering. Cambridge: Cambridge University Press, 2010. 1–31. Print.

[3] NVIDIA. “NVIDIA Professional Graphics Solutions.” NVIDIA, 2024, https://resources.nvidia.com/en-us-design-viz-stories-ep/l40-linecard?lx=CCKW39&&search=professional%20graphics. Accessed 26 March 2024.

[4] AMD. “AMD Ryzen™ Threadripper™ Processors.” AMD, 2024, https://www.amd.com/en/products/processors/workstations/ryzen-threadripper.html#tabs-705187c2a6-item-19a50af67b-tab. Accessed 26 March 2024.

[5] Liu, Yongjun, et al. ‘Accelerating the Simulation of Finite Difference Time Domain (FDTD) with GPU’. 2021 IEEE International Joint EMC/SI/PI and EMC Europe Symposium, 2021, pp. 707–711, https://doi.org/10.1109/EMC/SI/PI/EMCEurope52599.2021.9559260.

[6] Weiss, Alec, et al. ‘Acceleration of FDTD Code Using MATLAB ’s Parallel Computing Toolbox’. Advances in Time-Domain Computational Electromagnetic Methods, John Wiley & Sons, Ltd, 2022, pp. 453–489, https://doi.org/10.1002/9781119808404.ch12.

[7] Warren, Craig, et al. ‘A CUDA-Based GPU Engine for gprMax: Open Source FDTD Electromagnetic Simulation Software’. Computer Physics Communications, vol. 237, 2019, pp. 208–218, https://doi.org/10.1016/j.cpc.2018.11.007.

[8] Badía, José M., et al. ‘GPU Acceleration of a Non-Standard Finite Element Mesh Truncation Technique for Electromagnetics’. IEEE Access, vol. 8, 2020, pp. 94719–94730, https://doi.org/10.1109/ACCESS.2020.2993103.

[9] Arduino, Alessandro, et al. ‘GPU-Accelerated Finite Element and Finite Difference Methods for Scattering Problems in Voxel-Based Human Models’. 2021 XXXIVth General Assembly and Scientific Symposium of the International Union of Radio Science (URSI GASS), 2021, pp. 1–4, https://doi.org/10.23919/URSIGASS51995.2021.9560241.

[10] Schneider, John. “School of Electrical Engineering and Computer Science.” Understanding the Finite-Difference Time-Domain Method, https://eecs.wsu.edu/~schneidj/ufdtd/ufdtd.pdf. Accessed 3 June 2024.

[11] “EuroCC National Competence Center Sweden.” GPU Programming: When, Why and How?, https://enccs.github.io/gpu-programming/. Accessed 3 June 2024

[12] Jain, Sandeep. “Thread in Operating System.” GeeksforGeeks, 26 February 2024, https://www.geeksforgeeks.org/thread-in-operating-system/. Accessed 6 June 2024.

[13] “NVIDIA CUDA Programming Guide.” NVIDIA, 16 April 2012, https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed 6 June 2024.

[14] “EuroCC National Competence Center Sweden.” Introduction to GPU architecture, https://enccs.github.io/openmp-gpu/gpu-architecture/. Accessed 6 June 2024.

[15] Peng, Wang. “Nvidia.” Fundamental Optimizations in CUDA, https://developer.download.nvidia.com/GTC/PDF/1083_Wang.pdf. Accessed 3 June 2024.

[16] “NVIDIA GeForce RTX 3070 Specs.” TechPowerUp, https://www.techpowerup.com/gpu-specs/geforce-rtx-3070.c3674. Accessed 3 June 2024.

[17] Schneider, John. “School of Electrical Engineering and Computer Science.” Understanding the Finite-Difference Time-Domain Method, https://eecs.wsu.edu/~schneidj/ufdtd/ufdtd.pdf. Accessed 3 June 2024.

8 APPENDIX

8.1 Memory coalescence

To prevent stalling, values required by threads in a warp should be placed adjacent in memory. This task is especially non-trivial due to the offset E and H grids that have alternating access patterns. Figure 5 shows an example of a basic blocked arrangement.

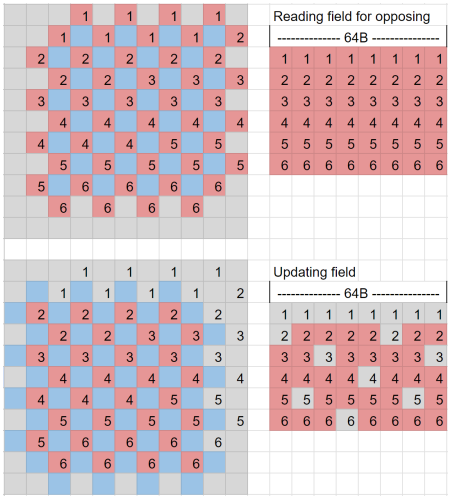

Figure 5: Basic memory layout for coalescence

This arrangement provides perfect coalescence when reading the field for use in the opposite update. This allows for full utilisation of L1 cache by only loading useful information. When updating it can be seen that there are several unused values. When considering an adjacent warp it becomes far worse. The adjacent warp would require two times the memory accesses due to being unaligned with the blocks. The following design is far more considerate of adjacent warps.

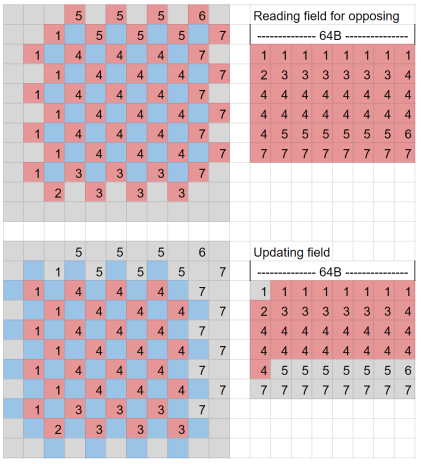

Figure 6: Better memory layout for coalescence

It places the values from shared borders adjacent in memory. Adjacent warp access now does not become unaligned on the overlapping regions. This comes with added complexity whenever indexing the memory.

For the rest of the design, the better layout will be considered. Thus reading the opposing field 3 memory accesses. Updating a field requires 6 accesses (read and write). It requires 6 instead of 4 as the coalesced access must be block-aligned.

8.2 Warps per block/SM

In this section lower and upper guidelines for the number of warps per block will be explored. This information is not meaningful when working with small simulation sizes.

All SMs must be used to achieve maximum performance. This requires at least 1 block for each SM. Using the metric of 25 warps per block that will be found in 8.2.1, at least 36800 electric grid points are required to avoid unutilised resources. With the standard 20 grid points per wavelength and 30MHz chosen this requires a simulation area of at least 9200 m^2. Reducing simulation size would likely take less time to compute but be decreasingly efficient.

8.2.1 Lower guideline

The lower guideline refers to the minimum number of blocks to mask any required global memory access. To calculate this the cycles taken for arithmetic operations are required. A magnetic field update requires the least arithmetic with 2 multiply and 2 add/subtract operations on doubles. It also has 3 multiply and 7 add operations on integers. The number of clocks these operations take can be calculated from GPU specifications. The integer operations will be ignored for simplicity as they are negligible compared to operations on doubles.

A 3070 has a theoretical 317.4 GFLOPs for doubles [16]. FLOPs, floating-point operations per second, measure the multiply addition performance with one multiply and one add equivalent to 2 FLOPs. With 46 SMs that is 46 * 4 = 184 warps. Therefore each warp is processing ~1.725 GFLOPs each. With a boost clock of 1725 MHz, 1 warp can do 1 FLOP per clock. Each thread requires 4 FLOPs resulting in 128 FLOPs per warp at a minimum for FDTD updates.

According to NVIDIA, global memory access on the GPU takes 400-800 cycles. It will be assumed that due to bussing and

coalescence, it will take 800 cycles regardless of the number of accesses. If one warp is accessing memory for 800 cycles it requires 6.25 other warps available to prevent stalling. That is assuming continuous operation however and when starting there is still ~800 cycles of stall time for the first executed warp to receive its data. Due to this reason, it is still beneficial to have more warps available for each warp unit. However, at a minimum, there should be 6.25 warps per warp unit (25 warps per SM). As warps are not assigned specific warp units that value is intentionally not rounded until the block / SM level. Following the same process electric field updates require at least 17 warps per SM.

8.2.2 Upper guideline

The upper guideline is how many active warps a block can have before potential L1 Cache capacity misses.

Updating an electric field point requires access to 4 magnetic field points and the updating point. This is more than those required for a magnetic update. A square warp simplifies tiling and provides good reuse of values between threads. The 32 threads updating 32 electric field points require 48 unique magnetic field points, see Figure 7.

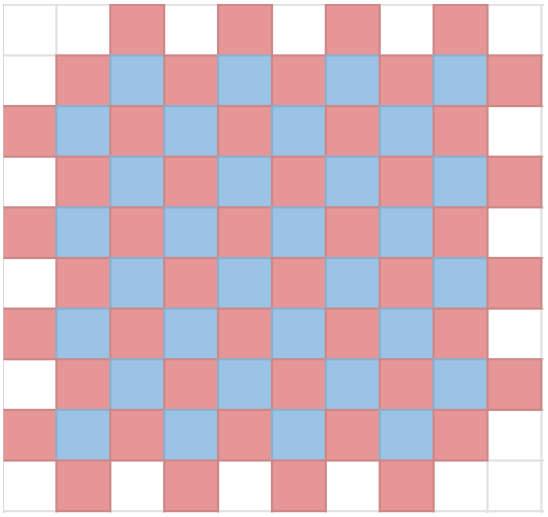

Figure 7: Magnetic field points (red) required by warp updating electric field points (blue)

Following this pattern the number of distinct magnetic field points required for n by m warps in a single rectangular block is

$$H_{xy} = 32nm + 8(n + m)$$

This can be used to estimate the cache space used by a block with n by m warps.

$$Cache\ Space = 8(64nm + 8(n + m))$$

$$= 64(8nm + n + m)\ \text{Bytes}$$

This estimate assumes grids are rearranged to perfectly place values within a block adjacent to one another. Equating it to the L1 Cache size of 128KB and rearranging for the m yields.

$$m_{max}(n) = floor(\frac{2000-n}{8n+1})$$

Graphing this function (black points) as well as the line of 128KB is shown in Figure 8.

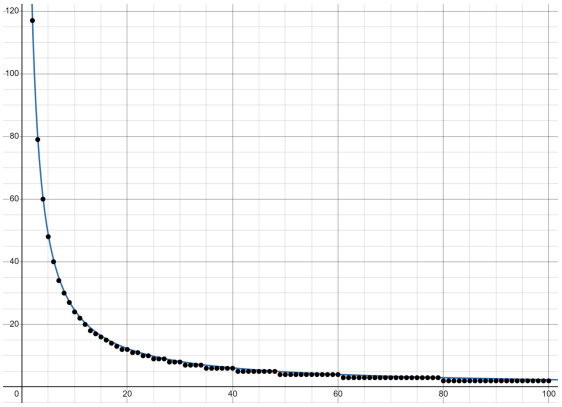

Figure 8: Maximum m warps (Y) with n warps (X) at the limit of 128KB cache usage.

To see how the dimensions may limit the maximum number of warps within a block the corresponding warp count is graphed in Figure 9.

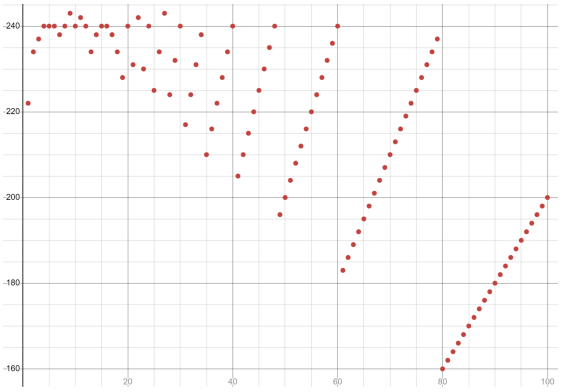

Figure 9: N dimension by Max warp count

200+ warps per block are easily achieved within L1 Cache. This allows the same efficiency at simulations beyond 73600 m^2 with 30MHz and 20 points per wavelength. This is significantly higher than the lower guideline providing great flexibility in the implementation.

In case the L1 Cache does not operate as assumed here and values cannot be maintained in the cache, a similar process can be used with the SMs' shared memory which is similar in size.

8.3 Implementation details

The first cycle works on implementing FDTD in one dimension. This is very simple as there is little to change from the FDTD reference implementation from John B. Schneider [17]. The animation code is implemented by exporting simulation data and importing it into Matplotlib, Python. Debugging is then done by creating animations and checking visually for any errors, distortions, or artifacts.

A similar process is then followed for the 2D implementation. However, none of the supporting code could be reused. The 2D implementation had far more supporting code with the different ABCs and sources. Thus trying to overlay it on the single file 1D implementation would become too messy. The animation now required a heatmap instead of a scatter plot. Debugging could then be done.

The animator is far slower than expected for small grid sizes (250 x 200). The animator is intended to also verify far larger grid sizes in the GPU-accelerated implementation. The slowness may have been from plotting overheads, not data size. Thus it is logical to attempt animation of a significantly larger but still relatively small grid size (19500 x 883). At this size the animator is unusable. After some research, it is determined that using Julia for animation would provide a similar interface but be far faster. After porting the code to Julia, the animations are fast enough.

Now code to load realistic terrain data into the simulation is created. It involved modifying the implementation, creating an extra Python script, and manually tracing an elevation profile.

Up to this point the system had been designed for modularity. Provided they followed the interface, this design allowed for modular sources, ABCs, terrain data, and samplers. It also had quality-of-life features like pausing and resuming simulation with varying sampling rates. However, while useful, this design is unfavourable for multithreading and didn’t provide enough meaningful benefits for this project. As such, the design is greatly simplified for the multi-threaded implementation.

Finally, the implementation is ready to be multi-threaded in preparation for GPU acceleration. However, as performance data will later be required of this version compared to later versions it is logical to implement the testing framework at this stage. Notably, a single-threaded and multi-threaded implementation running one thread is not equivalent. The only issues encountered regard variable types while computing output values resulting in overflow and underflow.