From the holdings of the Department of Special Collections , Princeton University Library, Princeton, NJ

These documents are either in the public domain, or can only be used for educational and research purposes (Fair use) as per U.S. Copyright law (text below). By accessing this file, all users agree that their use falls in the public domain or within fair use as defined by the copyrightlaw. We do not charge any permission or usage fees for the publication of images of material inour collections, including those provided by Princeton University Library, via our website, catalog, or directly from staff.For more information regarding copyright, credit, and citation please review the information onthe Copyright, Credit and Citation Guidelines page of our website.
https://library.princeton.edu/special-collections/policies/copyright-credit-and-citation-guidelines. If you have any questions, including how to order higher-quality images from the collection, contact Special Collections Public Services via our Ask Us! Form. https://library.princeton.edu/special-collections/ask-us

---

Master's Theses, 2022-2026

Computer Science, 2023

2025

GPU Acceleration of Adaptive Mesh Refinement via a Hashtable

Khotiaintseva, Nataliia

Princeton, NJ : Princeton University

---

http://arks.princeton.edu/ark:/88435/dsp01mp48sh20j

U.S. Copyright law: The copyright law of the United States (Title 17, §108, United States Code) governs the making of photocopies or other reproductions of copyrighted material. Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be used for any purpose other than private study, scholarship, or research. If a user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of fair use that user may be liable for copyright infringement. This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve a violation of copyright law. https://www.copyright.gov/title17/title17.pdf

GPU ACCELERATION OF ADAPTIVE MESH REFINEMENT VIA A HASHTABLE

Volume I

Nataliia Khotiaintseva

A MASTER'S THESIS

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF MASTER OF SCIENCE IN ENGINEERING

RECOMMENDED FOR ACCEPTANCE BY

THE DEPARTMENT OF

COMPUTER SCIENCE

Adviser: Romain Teyssier

May 2025

© Copyright by Nataliia Khotiaintseva, 2025.

All rights reserved.

ABSTRACT

In this work we explore acceleration of computations in AMR for astrophysical simulations. Modern scientific codes widely use Adaptive Mesh Refinement, and they need to leverage GPU supercomputing power. Specifically, this thesis demonstrates a hashtable implementation on GPU to store the grid structure. On a toy problem, we compare the performance of a typical matrix implementation against a hashtable implementation in a worst-case scenario — when all grid cells are uniformly refined. Acceleration of such large-scale astrophysical hydrodynamics simulations is critically important, making optimization techniques such as GPU-based hashtable storage essential. Our result is that there is no disadvantage in the hashtable implementation versus the matrix implementation. This result is crucial for future implementation of a hashtable-based AMR with dynamic refinement and complete hydrodynamics simulation. We also explore the application of the Grace Hopper superchip to AMR code and observe a significant speedup of ~2.3 in gradient calculation, and ~14.7 speedup in hashtable creation. Additionally, we design a linter program to streamline CUDA C++ debugging process, in particular for our AMR implementation. These approaches together aim to speed up scientific code; they combine large-scale scientific computing with cutting-edge tools in computer science.

1 INTRODUCTION

This thesis builds on the previous efforts to speed up Adaptive Mesh Refinement simulations for matter dynamics using GPUs. The original prototype of the underlying CUDA C++ code was developed as part of the COS 521 group final project, titled "GPU-Acceleration of Adaptive Mesh Refinement Algorithms" (Andalman, Z. et al. 2023). Further, COS 597E final project (Khotiaintseva, N. 2024) explored the linter program and a performance assessment on a GPU superchip Grace Hopper. Contributions from the course projects supplement the primary focus of this thesis and are included as personal additions to the topic.

We significantly advance prior work, summarizing three key contributions:

- AMR GPU acceleration using a hashtable to store the grid structure
- development of a static linter program to detect warp divergence issues in CUDA C++ code, improving debugging efficiency and implicitly enhancing performance of GPU programs
- a performance evaluation on NVIDIA Grace Hoper GH200, observing notable speedups for computations and memory allocation on GPU.

Together, these efforts address critical challenges in GPU-based astrophysical simulations and pave the way for more efficient large-scale modeling of fine-scale structures like supernova shocks (Figure 1) or accretion disks around black holes (Biernacki et al. 2017).

Our AMR GPU code with the linter program are available on GitHub[1].

1.1 AMR Background

AMR is a main focus of our work. It allows fine-grained resolutions in the interest areas of phenomena in space like supernova shocks. By implementing AMR on GPU, we strive to utilize its parallel computational power, which we demonstrate in our experiments on NVIDIA A100 and Grace Hopper GPUs.

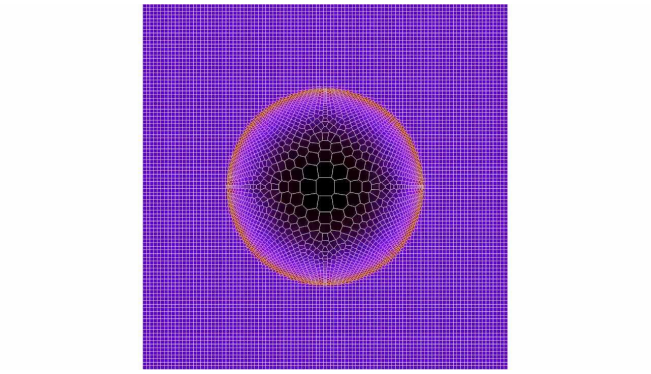

Figure 1. Sedov blast wave

Adaptive Mesh Refinement is typically a 3D grid with a higher resolution at the areas of interest. In astrophysics, AMR models the expansion of a strong shock wave (Figure 1) of gas and plasma from a supernova explosion. This expanding shock wave is called Sedov explosion and is used as a test scenario in `RAMSES` (Teyssier 2002) and its corresponding mini-ramses[2] codebase.

The structure of Adaptive Mesh Refinement is essentially a Cartesian grid with different cell sizes, for example smaller cells in the areas of interest. Adaptive Mesh Refinement code simulates dynamics of matter. It computes hydrodynamics of each cell in the grid relative to the cell's neighbors. In particular, mini-ramses solves partial differential equations under the physical conservation laws (Godunov 1959). The algorithm then decides how the values in each cell change in a time step. Based on the values it refines the most interesting areas of the grid by dividing cells into higher resolution or unrefine cells that are no longer interesting.

1.2 GPU and CUDA Background

Computers normally have a Central Processing Unit and a Graphics Processing Unit, which are the two main types of processors. We can see CPU's work in the Task Manager on Windows or via the `top` command on Mac or Linux. While a CPU is well optimized for everyday tasks that operate sequentially, a GPU deals with highly parallel tasks. It was originally designed for rendering images, but now is widely used in AI field, scientific computing and

[1] https://github.com/314152718/amr-gpu/tree/main

[2] https://bitbucket.org/rteyssie/mini-ramses

simulations due to its parallel processing design. We often see GPU power applied in gaming, machine learning, scientific simulations, and high-performance computing. The CPUs have few cores optimized for sequential tasks, while GPUs have thousands of smaller, efficient cores optimized for parallel tasks.

GPU architecture includes streaming multiprocessors: groups of cores that execute instructions in parallel. Within the cores, blocks of threads are located; within each block, the threads are grouped into warps (32 threads on NVIDIA GPUs). Such warps are the basic units of execution.

Our work, building upon the prototype in Andalman, Z. et al. (2023), is inspired by Fortran 90 mini-ramses[3]. We consider a toy problem of mini-ramses hydrodynamics calculation and compute the gradient of density ρ, stored in each cell of the grid. We accelerate this calculation on GPU, and utilize a GPU-based hashtable `cuco::static_map` that was recently developed by NVIDIA in 2021 as a part of `cuCollections` module[4] (Juenger et al. 2023).

2 METHODS

Our 3D AMR grid structure consists of individual cells in a cubic space $x, y, z \in [0, 1]$. We use a cell index scheme `(i,j,k,L)` from `RAMSES` (Teyssier 2002). L represents a refinement level starting from $L = 0$. The width of the cell's face is $\frac{1}{2^L}$. In other words, at $L = 0$ the only cell spans the whole cube $[0, 1] \times [0, 1] \times [0, 1]$. At the next refinement level $L = 1$, the parent cell at $L = 0$ can be divided equally into 8 cells. The cells are further divided according to the grid structure. The parent and child cells are stored in the grid. Similar to the cells' index representation in `RAMSES`, an index `(i,j,k)` represents the cell's position in a refinement level, where i, j, k are non-negative integers. For example, $i = 0$ represents the leftmost cell on the x axis. Under this scheme the children of a cell with index `(i,j,k,L)` are given by coordinates $[2i(+1), 2j(+1), 2k(+1), L+1]$, and the parent of a cell is represented by coordinates $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor, \lfloor k/2 \rfloor, L-1)$.

[3] https://bitbucket.org/rteyssie/mini-ramses
[4] https://github.com/NVIDIA/cuCollections

The following example explains the meaning of cell index coordinates (i, j, k). Starting at $L = 0$ there is one cell that spans the whole cube $[0, 1] \times [0, 1] \times [0, 1]$. This cell has index coordinates $(i, j, k) = (0, 0, 0)$. At the next level $L = 1$ a cell can be evenly divided into 8 smaller cells that would have coordinates ranging from $(0, 0, 0)$ to $(1, 1, 1)$. A grid of level L is **fully refined** if all possible cells are present at all levels 0 to L.

`RAMSES` includes a fully featured gas hydrodynamics solver on a grid. As such, it employs Godunov's scheme to solve partial differential equations governed by physical conservation laws (Godunov 1959). In Godunov's scheme, the gradient of conserved quantities is calculated in order to further compute fluxes on the cells' faces. Specifically, `RAMSES` calculates the gradients of pressure, density and Mach number. Pressure and density gradients are utilized to keep track of and refine the areas with steep changes in pressure or density, while Mach number gradient targets regions with varying flow velocity relative to the sound speed.

Within our simplified model we only compute gradient of density ρ on the faces without further hydrodynamics calculations. We consider that ρ value is constant across one cell and equal to ρ at its center coordinate.

2.1 Hashtable Structure

Our grid structure is stored as a map from a cell spatial index `(i,j,k)` and refinement level `L` to the cell's density value ρ. Within the `cuco::static_map` hashtable, collisions are handled using default `cuco::static_map` linear probing scheme. Under this rule the hashtable probes subsequent indices up to a free cell when it hits an occupied index. On the other hand, the standard `std::unordered_map` uses a separate chaining method, where colliding entries are stored as a linked list at the index they conflict with. Linear probing scheme for `cuco::static_map` is used because it can be well optimized for parallel threads on GPU. Specifically, multiple consecutive threads within the same warp fetch a block of indices in memory, and in parallel determine which index is free to place the entry into. On the other hand, separate chaining would introduce pointers that are inefficient for GPU memory access patterns.

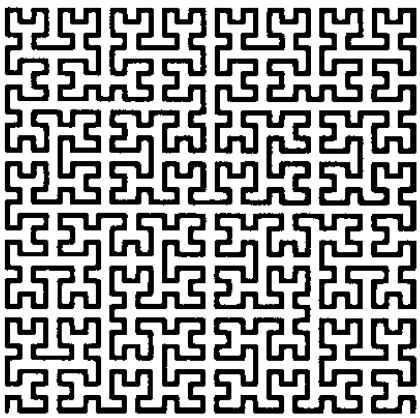

Figure 2. Hilbert space-filling curve

In the `cuCollections` library we are provided with `linear_probing` and `double_hashing` scheme options. We tested `double_hashing`, which is designed to reduce clustering and allow for faster data placement; however, the performance was slightly slower. `double_hashing` works by using a second hash function provided to search for a free cell, instead of looking for the nearest neighbor. The performance slowdown can be attributed to the need for neighboring cells to be located close to each other for efficient memory access, as discussed in the next subsection on Hilbert Index.

For our `cuco::static_map` hashtable implementation on GPU, we use the same hash function as outlined in `RAMSES` and applied in Andalman, Z. et al. (2023):

$$\text{hash}(i, j, k, L) = -1640531527i + 97j + 1003313k + 5L$$

This hash function is based on the prime numbers, and is specifically designed to have fewer collisions for typical AMR sizes. Its performance on GPU is somewhat better than `cuco::default_hash_function`. The timing comparison of these hash functions is provided in Experiment Setup and Results section (Figures 19, 20 and 21).

2.1.1 Hilbert Index

A certain performance improvement can be achieved by incorporating Hilbert space-filling curve (Figure 2) index as it is used in `GAMER-2` paper (Schive et al. 2018) and as was attempted in the COS 521 project (Andalman, Z. et al. 2023). On a GPU we store the underlying `(i,j,k,L)` *keys* array and ρ *values* array, and also the `cuco::static_map` hashtable for fast cell lookup given its index. Gradient calculation at the cells' faces requires accessing ρ values of their neighboring cells. GPU by its nature accesses memory in chunks, so the gradient calculation can be optimized if the neighboring cells are stored nearby in memory. If the nearby positioning is achieved for the *values* array, memory chunks would be fetched on GPU fewer times when a neighboring cell is accessed in the *values* array. Such neighbor positioning can be achieved by storing *keys* and *values* arrays using Hilbert index order. It ensures that the neighboring cells are likely to be stored close together in GPU memory as well. Such optimization accelerates gradient calculations by taking advantage of the cache locality. Cache misses are minimized when the neighboring cells are fetched in memory consecutively. Moreover, Hilbert order improves GPU memory coalescence, reducing the number of transactions needed for lookups. Furthermore, it follows that the warp execution is more efficients because the threads in the same warp are likely to access neighboring memory locations. This topic is further explored in the Performance Optimization Techniques section.

Cells in *keys* and *values* arrays are ordered by their refinement level. Hilbert curve cell ordering would imply that *keys* and *values* for a refinement level L are ordered using L-th order Hilbert curve. Given an index (i, j, k, L), we can compute the corresponding Hilbert index using the algorithm detailed in Skilling (2004). As a result, the total index in *keys* and *values* would be equal to Hilbert index at the level L plus the sum of the total number of cells at fully refined levels $l < L$:

$$\sum_{l=0}^{L-1} 2^{3l} = \frac{2^{3L} - 1}{2^3 - 1}$$

2.2 Grid Density Initialization

In the previous work (Andalman, Z. et al. 2023) our code was first implemented on CPU, where we verified the correctness of the gradient calculations. The example field used for the matter density was Gaussian. All cells were first initialized with the Gaussian value ρ

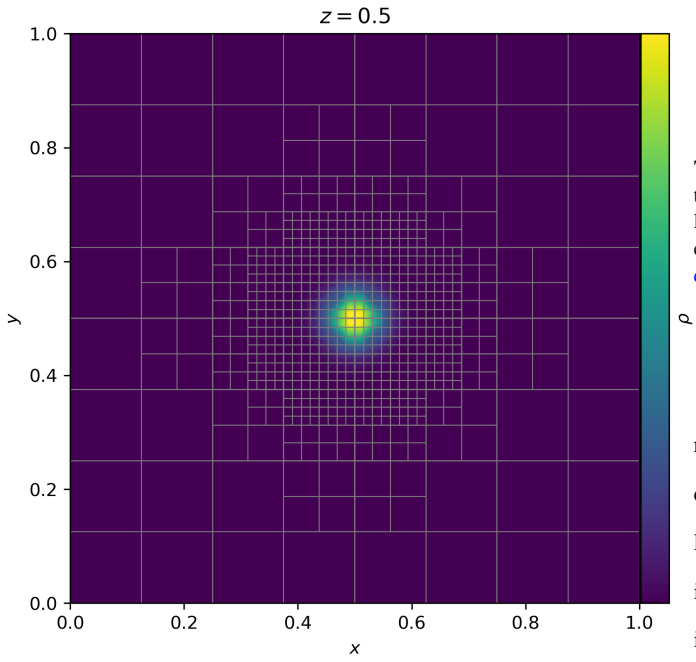

Figure 3. Gaussian cell refinement with levels 3 to 6

using Gaussian formula:

$$\rho(x,y,z) = \frac{1}{\sigma^3(2\pi)^{1.5}} \cdot e^{-\frac{(x-x_0)^2+(y-y_0)^2+(z-z_0)^2}{2\sigma}}$$

Here $x_0 = y_0 = z_0 = 0.5$, which is a coordinate of the center of our cubic domain $[0,1] \times [0,1] \times [0,1]$. We also set $\sigma = 0.01$ in our example. The Gaussian distribution was chosen because its gradient can easily be verified using this formula:

$$\nabla\rho = -\frac{\rho}{\sigma}\left[(x-x_0)\hat{\mathbf{x}} + (y-y_0)\hat{\mathbf{y}} + (z-z_0)\hat{\mathbf{z}}\right]$$

We start with a fully refined grid at level $L = 3$, so the cube grid $[0,1] \times [0,1] \times [0,1]$ is evenly divided into cells of size $\frac{1}{2^3} = \frac{1}{8}$ as portrayed in a Figure 3. Although our grid is coarse and the gradient cannot be as precise as the real Gaussian field gradient, the residuals between numerical gradient and the theoretical field gradient were computed to be less than 1%.

2.3 Refinement

We adopt a simple refinement criterion $\rho > \rho_{crit}$, where ρ_{crit} is set to 0.01, as was done in Andalman, Z. et al. (2023). The resulting circle refinement boundary is visible in Figure 3. According to a refinement rule from `RAMSES`, a refinement level of neighboring cells cannot differ by more than one. As a result, if a cell with a refinement level L has a bigger neighbor with level $L-1$ and needs to be refined, its bigger neighbor must be refined as well. This explains the resulting image shown in Figure 3.

Left refin. level	Right refin. level	a	b	c	d
L	L	-1	0	1	2
L	L-1	-9	5	4	15
L-1	L	-4	-5	9	15
L-1	L-1	-1	0	1	3

Table 1. First-order finite difference coefficients. The coefficients depend on the refinement of the left and right neighbors, where our cell has refinement level L. Left refinement level indicates refinement level of the left neighbor etc. This table of coefficients was used in COS 521 project (Andalman, Z. et al. 2023).

In our previous work we have implemented a recursive cell refinement algorithm:

```
# ---pseudo code---

# cell is at a level L
def refineCell(cell):
    children = addChildrenToGrid(cell)
    for direction in 3dDirections:
        #neighbor at level L-1
        neighbor1 = getNeighborCoord(
            cell, direction, cell.L-1
        )
        if not exists(neighbor1):
            #neighbor at level L-2
            neighbor2 = getNeighborCoord(
                cell, direction, cell.L-2
            )
            refineCell(neighbor2)
```

For this Gaussian density example we pick a maximum refinement level $L = 6$. To achieve the refinement pattern shown in Figure 3, we apply the recursive `refineCell` algorithm to each cell that meets the refinement criterion $\rho > \rho_{crit}$.

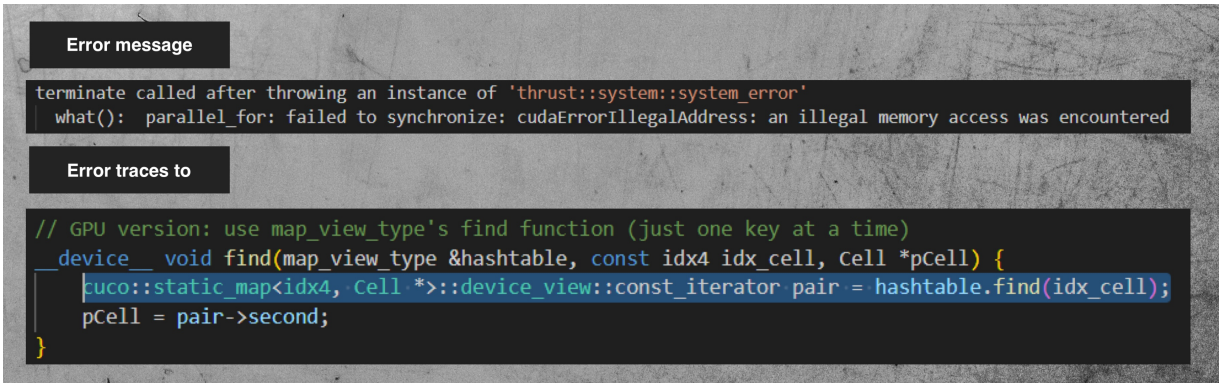

Figure 4. Runtime error message on GPU for `cuco::static_map` hashtable

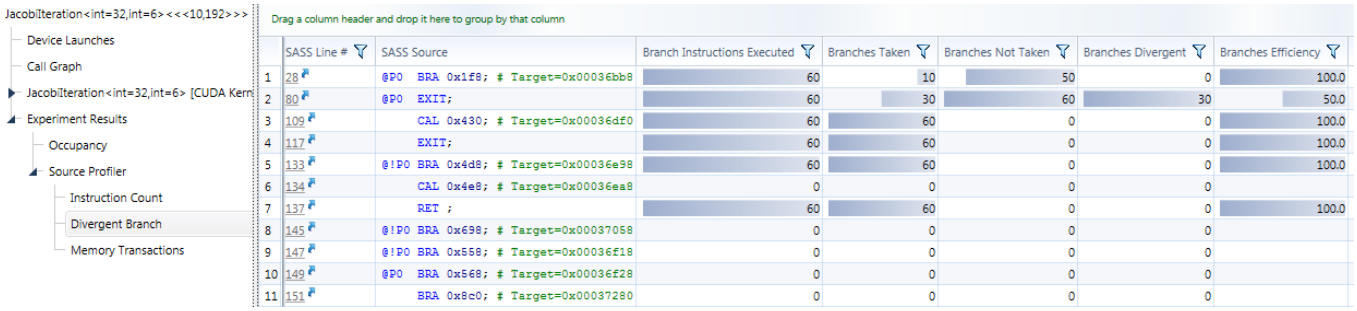

Figure 5. Divergent Branch experiment

```
for (int i = 0; i < repeat; i++) {
    auto start = high_resolution_clock::now();

    // run as kernel on GPU

    calcGrad<<<grid_size, block_size>>>(find_ref,
                                        insert_keys_it,
                                        num_inserted[0]);
    cudaDeviceSynchronize();
    CHECK_LAST_CUDA_ERROR();
```

Figure 6. Checking for GPU runtime error after a GPU function call

```
#define CHECK_LAST_CUDA_ERROR() checkLast(__FILE__, __LINE__)

void checkLast(const char* const file, const int line)
{
    cudaError_t const err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Runtime Error at: " << file << ":" << line
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        // We don't exit when we encounter CUDA errors in this example.
        // std::exit(EXIT_FAILURE);
    }
}
```

Figure 7. Definition of `CHECK_LAST_CUDA_ERROR()`

2.4 Gradient Calculation

We use coefficients to compute the gradient approximation in a cell depending on the refinement level of its left and right neighbors. These coefficients are sourced from Commerçon et al. (2011) and applied in Andalman, Z. et al. (2023). We store parent cells from every previous level in our grid, and by a convention refinement levels of the leaf cells differ at most by one. Therefore, we can access either a neighbor of the same refinement level, or one level below. These cases for the left and right neighbors are presented in the Table 1. An exception case would be a border where we simply define the gradient to be 0. This boundary value in fact matches our Gaussian example density field.

We are using the following formula to compute the approximate gradient, consistent with the approach from our previous work:

$$(\nabla\rho)_x^{(\text{center})} \approx \frac{a\rho^{(\text{left})} + b\rho^{(\text{center})} + c\rho^{(\text{right})}}{d\Delta x}$$

Coefficients mentioned in Table 1 can be explained by the neigh-

bor refinement levels. Bigger neighbor centers are located further apart from the central cell. Cells with lower refinement levels are spaced farther apart, so the neighbor influence on the gradient value is redistributed accordingly. Finally, the denominator d scales the coefficients to keep consistency with the gradient dimensions.

3 LINTER PROGRAM

3.1 Warp Divergence and Motivation

Developing CUDA C++ GPU code presents significant challenges. Warp divergence is one of the problems arising in GPU programming. It is a performance issue that happens due to `if-else` statements in the device code where threads within a warp take different execution paths. In this case the GPU scheduler executes all `if` branches **sequentially** first, followed by all sequential `else` branches, significantly reducing parallel efficiency. If there is no `else` statement, then the same divergence happens with the branches for which the `if` statement does not hold. However, when `if` statement always holds, there are no issues.

To avoid this issue, in `__device__` functions we add boolean variables and with their help rewrite the `if-else` blocks. For example (Andalman, Z. et al. 2023):

```
// subtract 1 from the AMR level
// if the neighbor is not refined
idx_neighbor.L = idx_cell.L - int(is_notref);
```

In order to detect warp divergence, the linter program developed in this project uses `clang.cindex` library in Python to traverse variable tree and to pinpoint the `if-else` blocks in `__device__` functions.

The intricacies of GPU programming often lead to inefficiencies and frustration. This linter is designed to simplify the process, enabling developers to write more efficient and reliable code. For example:

- Error messages on GPU may not be caught by a compiler, such as the runtime error `cudaErrorIllegalAddress` (Andalman, Z. et al. 2023) in Figure 4.
- Printing on GPU requires the use of `printf` instead of `cout <<`. Moreover, one of the inherent constraints of GPU programming is the inability of CUDA threads to perform file I/O operations (NVIDIA 2024a). The file output for debugging purposes needs to be done on CPU.
- CUDA C++ does not support `std::string` for GPU execution (NVIDIA 2024a).
- We can only see the runtime error in CUDA after GPU finishes all the work. To see the runtime error we need to call `cudaDeviceSynchronize()` after the asynchronous calls to GPU functions. It was then necessary to define our own function to output runtime errors in CUDA (see Figure 6 and 7)

Following these considerations, various linter programs would be useful for debugging.

3.2 Existing Tools for Warp Divergence Detection

There are some powerful existing tools for performance evaluation and even specifically for warp divergence detection. For example, NVIDIA Nsight Compute (NVIDIA 2024b) software provides Divergent Branch experiment (NVIDIA 2015). It examines every branch and counts the number of threads that took each branch (see Figure 5). However, while Nsight Compute performs runtime analysis, our linter program is a static tool designed to identify bugs at an earlier stage.

Another powerful runtime profiling tool is NVIDIA Nsight Systems (NVIDIA 2024c). It analyzes runtime, kernel performance and can be integrated with Nsight Compute, however, it does not provide specific analysis for warp divergence.

There are more tools available such as CUDA-GDB (NVIDIA 2024d) and Nsight Graphics (NVIDIA 2024e) for debugging, and CUDA Memcheck (NVIDIA 2024f) and Compute Sanitizer (NVIDIA 2024g) for memory analysis, but they are not specific either for warp divergence analysis.

Building on the limitations of existing tools for warp divergence detection, our linter program offers a static approach for identifying

issues earlier in the development cycle. The following subsection provides an overview of its implementation and underlying GPU code syntax.

3.3 GPU code syntax

Starting from CPU, the code first enters a GPU using the syntax as in Figure 10. Such functions have a `__global__` keyword prefix.

`BLOCK_SIZE` is the number of threads that form a block, and within the block threads are grouped in groups of 32 (warp). The total number of threads is equal to `BLOCK_SIZE x GRID_SIZE`. For `calcGrad` call I set the `BLOCK_SIZE` to 512, as changing it did not cause significant difference for `calcGrad` timing. We then set `GRID_SIZE = (num_cells + BLOCK_SIZE - 1)/BLOCK_SIZE` so that the number of threads is lower bounded by the number of cells in a grid's current refinement level L.

Within the `__global__` function `calcGrad` (Figure 9) the GPU-wide thread id is calculated as `tid = threadId.x + blockId.x * blockDim.x`. Here, `threadId, blockId` and `blockDim` are predefined CUDA variables. Each thread executes `calcGrad` simultaneously at first. Then if the number of threads happened to be less than the processed vector length `num_keys`, the `while` loop starts to be used. A flattened 1D version of cell indices `(i,j,k,L) contained_keys` array is then iterated uniquely for every `threadId`. Here a warp divergence can occur if there is an `if` statement in `calcGrad` that depends on `tid` or `contained_keys`. Since in `__global__` functions the main code can be expected to be in a loop, in the linter's output we can include the loop information for `if` statements (see Figure 8).

In the execution path, following `__global__` functions that are called from CPU, all the other GPU functions are typically labeled with a `__device__` keyword. These can be called from both `__global__` and `__device__` functions. In our example `__device__ calcGradCell` is called from the `__global__ calcGrad` to further calculate gradient in a cell.

3.4 Linter Implementation

Using `clang.cindex` library in Python, I traversed the variable tree in `amr-gpu.cu` file[5]. `clang.cindex` automatically parses the files and classifies the body content of the function as its children. All the keywords and variables also follow through in the tree traversal. However, there were a few significant limitations for `clang.cindex`:

- `clang.cindex` does not parse the body of `template<> __device__` functions (such as calcGrad at Fig. 9) treating those as standalone code chunks in curly brackets. The solution was to automatically preread the file and comment out all lines starting with `template` for linter purposes.
- `__device__, __global__` keywords are also sometimes not parsed as tokens of the functions. The solution was to check them at 5 earlier positions in the variable tree.

I used `clang.cindex.CursorKind.VAR_DECL` to identify variables, and `FUNCTION_DECL, FUNCTION_TEMPLATE` to identify functions. `"__device__"` and `"__global__"` were kept as GPU keywords in a `gpu_kw` variable that is passed to child nodes. When observing a loop during traversal (`clang.cindex.CursorKind.WHILE_STMT` or `FOR_STMT`) that received a `"__global__" gpu_kw` from parent nodes, we record these loops and pass them to child nodes in the tree traversal as `"for"` or `"while"` stored in a `gpu_loop_kw` variable. This way we can remember what `if` statements were found specifically in loops inside `__global__` functions and record this in the output (Figure 8). A more detailed output is available on GitHub[6], showing that all GPU functions were visited.

3.5 Linter Program Future Work

Currently my linter program successfully traces `if` conditions in GPU functions. Debugging and optimization would truly benefit if the linter could check dependency of `if` conditions on `threadId.x`. This would mean checking whether `if` conditions explicitly include

[5] https://github.com/3141527718/amr-gpu/blob/main/amr-gpu.cu
[6] https://github.com/3141527718/amr-gpu/blob/main/linter.ipynb

```
WARNING: potential warp divergence due to "if" statement at 223 in __device__ getNeighborInfo
WARNING: potential warp divergence due to "if" statement in a "while" loop at 336 in __global__ make1lvlGrid
WARNING: potential warp divergence due to "if" statement at 353 in __device__ setGridCell
WARNING: potential warp divergence due to "if" statement at 362 in __device__ setGridCell
WARNING: potential warp divergence due to "if" statement at 365 in __device__ setGridCell
WARNING: potential warp divergence due to "if" statement in a "while" loop at 443 in __global__ insert_vector_pointers
WARNING: potential warp divergence due to "if" statement in a "while" loop at 470 in __global__ insert
WARNING: potential warp divergence due to "if" statement in a "for" loop at 494 in __global__ printHashtable
```

Figure 8. Linter program's output

```
// compute the gradient
template <typename Map, typename KeyIter>
__global__ void calcGrad(Map hashtable_ref, KeyIter contained_keys, size_t num_keys) {
    auto tid = threadIdx.x + blockIdx.x * blockDim.x;
    contained_keys += tid;

    while (tid < num_keys) {
        idx4 idx_cell = *contained_keys;
        Cell *pCell = hashtable_ref.find(idx_cell)->second;
        calcGradCell(idx_cell, pCell, hashtable_ref);

        contained_keys += gridDim.x * blockDim.x;
        tid += gridDim.x * blockDim.x;
    }
}
```

Figure 9. `__global__` calcGrad function

```
auto find_ref = hashtable.ref(cuco::find);
calcGrad<<<GRID_SIZE, BLOCK_SIZE>>>(find_ref,
                                    contained_keys.begin(),
                                    num_inserted[0]);
```

Figure 10. calcGrad function call

`threadId.x`, or indirectly depend on it through a chain of variable assignments across several GPU functions.

Further enhancements can include an embedded app for VS code, that can help solve other GPU programming challenges such as non-coalesced array access. In a perfect GPU operation, each thread would access only its corresponding array index. This allows GPU to fetch memory in chunks less frequently. For example, the linter could check if `num_threads` evenly divides the lengths of input arrays that are processed with `threadId.x` in a GPU function. This issue is also related to the speedup that comes from Hilbert index use for AMR GPU, as discussed in the Hilbert Index subsection, Performance Optimization Techniques section and Experiment Setup and Results section (Figure 2).

4 PERFORMANCE OPTIMIZATION TECHNIQUES

The main performance improvement is achieved from using a GPU-based hashtable to handle grids with targeted refinement areas. Typical approaches to AMR storage include three main methods. First, AMR data may be stored on the CPU. Second, neighbor pointers can be stored in each cell. Third, a fully refined matrix may be used, which often results in storing unnecessary data. For example,

- `RAMSES` (Teyssier 2002) stores AMR on CPU.
- `GAMER-2` (Schive et al. 2018) uses GPU to speed up hydrodynamics. However, it transfers data sequentially from CPU to GPU for small parts of the grid, so the computation is not fully concurrent.
- Menshov & Pavlukhin (2020) is operating the AMR grid on GPU but is storing the neighbor pointers in each cell. It also requires non-direct cell lookup from the less refined parent cell to the child cell.

We achieved another performance improvement from using Hilbert space-filling curve index. Its application to AMR GPU code is described in the Hilbert Index subsection above, and the resulting speedup is outlined in Experiment Setup and Results section. Spatial locality optimization promises to be even more effective for larger scale computations due to the following reasons. First, our results show that the speedup increases with refinement level L. Second, at higher refinement levels as the memory increases, there is a need to use more than one GPU machine. Then the physical distance between certain parts of the hashtable *keys* and *values* arrays will increase, making efficient neighbor ordering crucial. We also observe that a Space-Filling Curve is used in `GAMER-2`, and Hilbert ordering specifically is applied in Menshov & Pavlukhin (2020).

5 EXPERIMENT SETUP AND RESULTS

Considering advantages of using a hashtable such as efficient key-value lookups, we used a matrix-based and a hashtable-based implementation for a fully refined grid slice at a single level L from 0 to 9. The original goal of the underlying research was to show

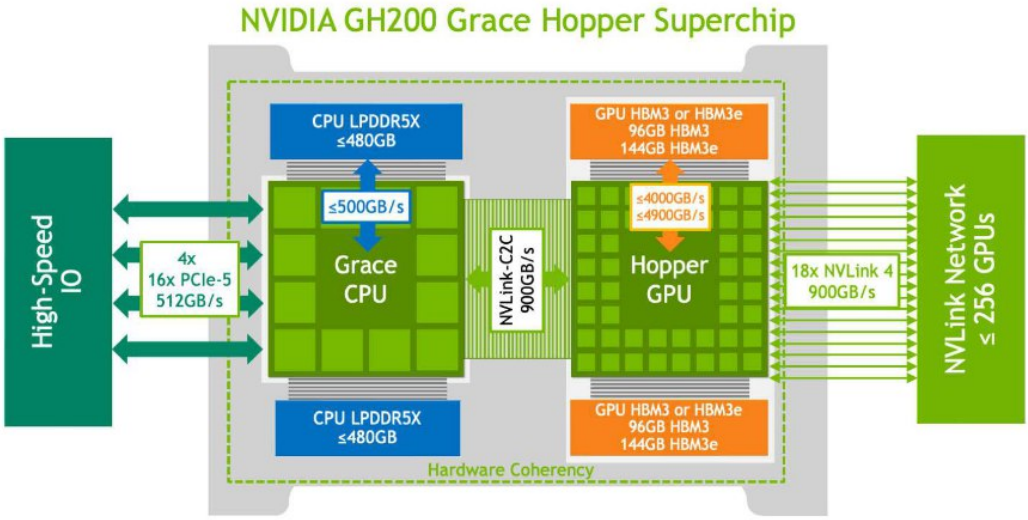

Figure 11. GH200 architecture scheme and specification (NVIDIA 2024h)

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity
** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Figure 12. A100 specification (NVIDIA 2021)

that a hashtable is nearly as efficient as the matrix for a fully refined level. This implies that using a hashtable offers significant advantages for handling intricate grid structures. The hashtable is implemented using `cuco::static_map` structure by `cuCollections` module[7] (Juenger et al. 2023), while the matrix implementation is flattened out in a 1D array of type `thrust::device_vector`. Serving as the main interaction point between the CPU and GPU, this vector automatically transfers data to the GPU when initialized on the CPU.

The hashtable size $1227133513 = 1 + 2^3 + 2^6 + ... + 2^{30}$ was selected to be a constant across all levels from 0 to 9, determined by the maximum number of cells at level 10. It was set constant across all refinement levels because `cuco::static_map` requires the hashtable size to be defined at a compile time. Next, the size was set to be one level higher than the maximum $L = 9$, as a hash function performs more efficiently when the hashtable is at least twice as large as required. Additionally, performance may improve slightly if the hashtable size is even larger than that.

5.1 Grace Hopper versus NVIDIA A100 GPU

A performance boost was explored from running the code on NVIDIA Grace Hopper GH200 superchip (NVIDIA 2024i) in com-

[7] https://github.com/NVIDIA/cuCollections

parison to NVIDIA A100 GPU. Grace Hopper prioritizes fast connections between GPU and CPU nodes, significantly accelerating data transfer—a common bottleneck in GPU programming. For GPU gradient calculation the speedup from using GH200 was ~2 times, while the above extent was ~14.7.

NVIDIA SXM A100 GPU on Stellar features Ampere architecture with 40GB of GPU memory[8]), while GH200 features Grace Hopper Architecture with up to 96GB of GPU memory (NVIDIA 2024h). Regarding the memory transfer speed between CPU and GPU, A100 interconnect memory bandwidth is reported to be 64GB/s (Figure 12) by links PCIe Gen4 that connect CPU and GPU, and GH200 interconnect memory bandwidth is 900GB/s (Figure 11). As a result, a theoretical speedup for `cuco::static_map` creation should be close to 14.1 times, which was confirmed by experimental speedup of ~14.7×.

5.2 Hashtable Creation Timing

Creating a hashtable of the above size consistently took ~28 ms on GH200, compared to 410 ± 13 ms on the NVIDIA A100, resulting in a 14.7× speedup for the GH200. This estimate is fairly accurate because it is close to the theoretical ratio 14.1. However, without the theoretical bound it would be difficult to argue about the accuracy due to the following reasons:

- A100 has smaller memory, resulting in possible memory fragmentation closer to the end of a program's execution. This could cause longer times for memory allocation.
- At times allocation can be very quick (a few ms) due to caching, which appears to occur more frequently for `thrust::device_vector` than for `cuco::static_map` in our program. This is why we cannot make straightforward conclusions about the timing differences of `thrust::device_vector` creation for the matrix implementation.

While `static_map` is a static structure, `device_vector` allows more dynamic actions, which explains why it could encounter caching more often.

[8] https://researchcomputing.princeton.edu/systems/stellar

5.3 Performance Graphs

In this work we ran both versions on an NVIDIA A100 GPU and on a Grace Hopper GH200 on Princeton's Stellar and Della research computing clusters respectively. Figures 13, 14 and 15 present the performance comparison of the gradient calculation. Figure 13 compares the actual speed, while 14 emphasizes how closely related the performance lines are for A100, GH200 matrix and hashtable implementations. For a better comparison Figure 15 highlights the ratio of A100 to GH200 performance speed. In Figure 13 the number of cells is 2^{3L} at a level L where 3 is the number of the grid's dimensions.

For all the graphs in this section each performance speed result is an average of 10 consecutive runs with an exclusion of outliers. Normally the first run in this case is notably slow for small refinement levels L.

Compared to the 14.7× speedup for hashtable creation, the gradient calculation was only ~2.3 times faster on GH200 for level $L = 9$. As shown in Figure 13, at $L = 9$ the gradient calculation took ~29 ms on GH200 while on A100 it took ~67 ms.

Due to Hilbert index curve, another speedup of ~2x was achieved at level $L = 9$. In Figures 16, 17 and 18 we see the performance comparison. Up to the level $L = 5$ the timing difference appears negligible, and starts being in favor of Hilbert index from $L = 6$. At the level $L = 9$ Hilbert index speeds up the gradient calculation from 69 ms to 35 ms.

We analyzed the hash function use and achieved a ~1.2x speedup at $L = 9$ from using the hash function presented in `RAMSES` (Teyssier 2002):

$$\text{hash}(i, j, k, L) = -1640531527i + 97j + 1003313k + 5L$$

The results are presented in Figures 19, 20 and 21. The speedup becomes not negligible from $L = 6$, as we can see in Figure 20. Before that the performance speed is close to a constant, and the lines closely interplay.

In Figure 14 we see that the log time for matrix and hashtable implementations on GPU is closely interwined, although on the last level $L = 9$ the matrix is ~2 times faster than the hashtable (13). As a result, the hashtable implementation may carry some overhead,

although the performance is almost at the speed level of the matrix implementation, considering the log time at other refinement levels L. This result for the hashtable was shown for a fully refined uniform grid structure. In a general case such as the expanding shock wave in Figure 1, the small cells in the grid are a lot more sparse. Therefore, we expect the hashtable implementation to demonstrate significantly better performance in practical cases where the grid is non-uniformly refined and small cells are sparsely distributed.

The matrix implementation has to store all possible cells (i, j, k, L) in the grid, even if they do not exist. However, the hashtable can avoid storing most of the nonexistent cells. The potential speedup from this fact is expected to be linear based on the number of stored cells. This can be confirmed by looking at the graph 14. Starting at $L = 5$ or $L = 6$, the performance graph is approximately linear. Since the number of total cells are exponential from L, this linear relationship on the graph is preserved when translating L into the number of cells, and performance log time into time. This is discussed in more detail in the AMR GPU Future Work subsection.

The execution time of the hashtable implementation is proportional to the number of cells. The execution time is comparable to that of the matrix implementation when the grid is fully refined. The full refinement means that the total number of cells is given by:

$$\sum_{l=0}^{L-1} 2^{3l} = \frac{2^{3L} - 1}{2^3 - 1}$$

If the maximum refinement level is L, the hashtable speedup can be expressed as $\frac{2^{3L}-1}{2^3-1}$ / `number_of_cells`.

In a target future implementation of these performance improvements for a complete hydrodynamics simulation such as mini-ramses[9], array creation would only happen once, while the gradient and other related calculations would be assessed each time step of the matter's flow in space. Therefore, we are more focused on the gradient calculation speedup results rather than the array creation.

I have experimented with `thrust::universal_vector` that utilizes unified memory in Grace Hopper and avoids data transfer, however, no noticeable performance improvement was observed in this case.

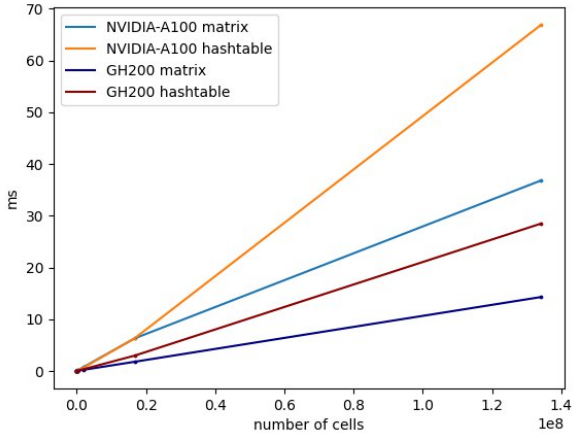

Figure 13. Gradient calculation performance

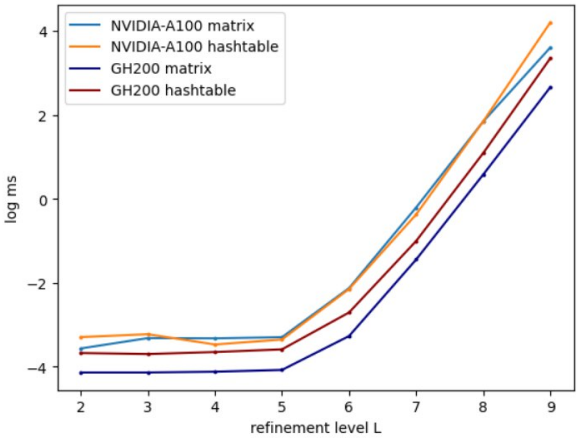

Figure 14. Gradient calculation performance with log axes

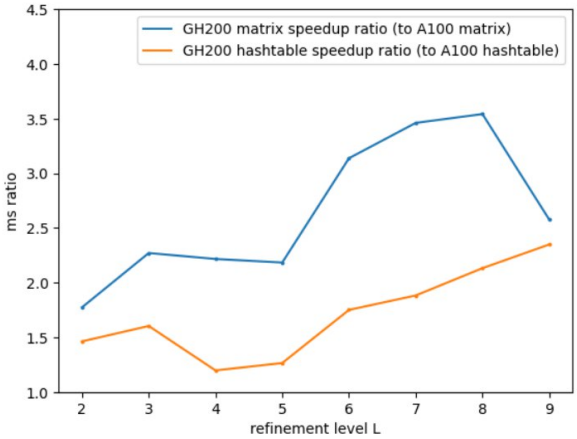

Figure 15. Gradient Calculation Performance speedup ratio (GH200 to A100)

[9] https://bitbucket.org/rteyssie/mini-ramses

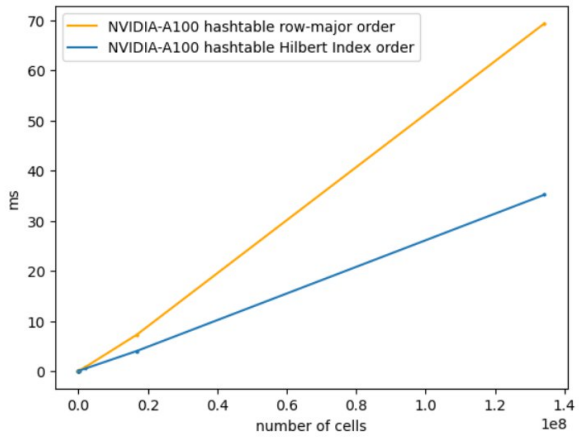

Figure 16. Gradient calculation performance

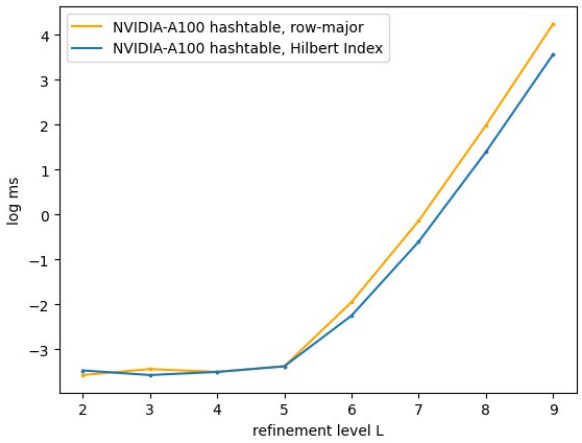

Figure 17. Gradient calculation performance with log axes

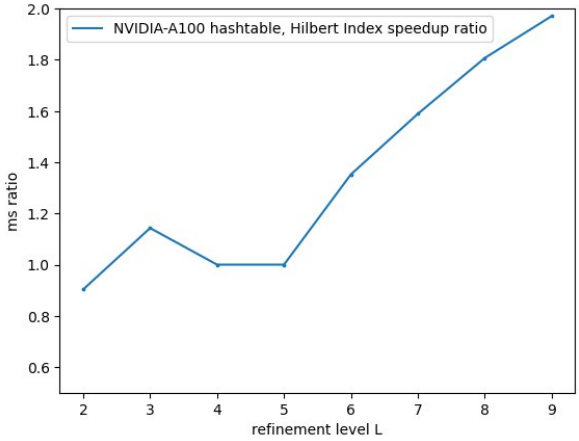

Figure 18. Gradient calculation performance speedup ratio on A100 with hashtable (Hilbert Index to row-major order)

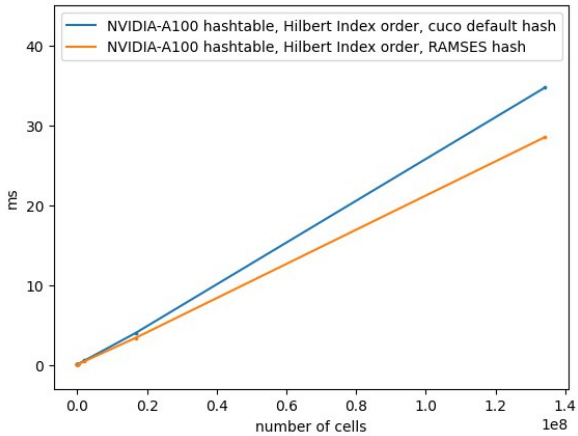

Figure 19. Gradient calculation performance

5.4 GPU Availability

Currently NVIDIA A100 instances are widely available, both for a single purchase[10] with 40GB or 80GB capacity per A100 node, as well as for multi-node setup purchase (8 instances or 640GB total)[11]. Yet, while GH200 is clearly accessible for educational purposes[12], it is only publicly available at the enterprise level configurations, such as 256 GH200 instances totaling 144TB[13]. Smaller-scale access has been announced on AWS EC2 instances, offering 32 GH200 instances (NVIDIA 2023). In addition, a faster chip Grace Backbell has been recently launched, and will be available on AWS in the form of GB200 NVL72, featuring 72 Backbell GPUs and 36 Grace CPUs (Amazon 2024).

5.5 AMR GPU Future Work

To highlight the hashtable's advantage for non-fully refined grids, future AMR GPU code research would include a performance comparison between matrix and hashtable implementations on grids with different refinement patterns, such as Gaussian (Figure 3). A key future challenge is integrating the GPU hashtable into hydrodynamics simulation codes, such as the Fortran 90-based mini-ramses[14].

It is important to note that currently we do not resize the *keys* and *values* arrays. However, in a complete hydrodynamics simulation as the matter keeps flowing, we need to constantly refine and unrefine

[10] https://www.amazon.com/NVIDIA-Ampere-Graphics-Processor-Accelerator
[11] https://viperatech.com/shop/nvidia-dgx-a100
[12] https://researchcomputing.princeton.edu/systems/della#Grace-Hopper-Superchip
[13] https://viperatech.com/shop/nvidia-dgx-gh200
[14] https://bitbucket.org/rteyssie/mini-ramses

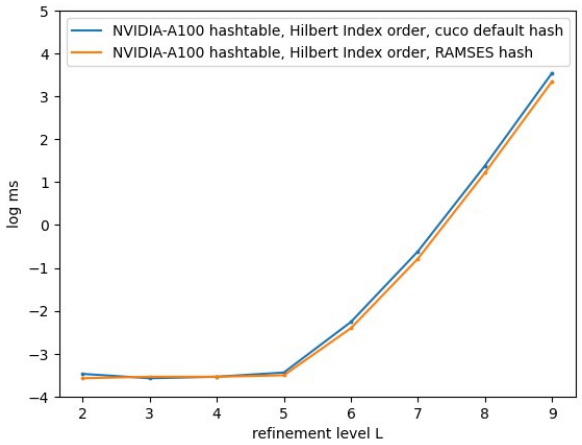

Figure 20. Gradient calculation performance with log axes

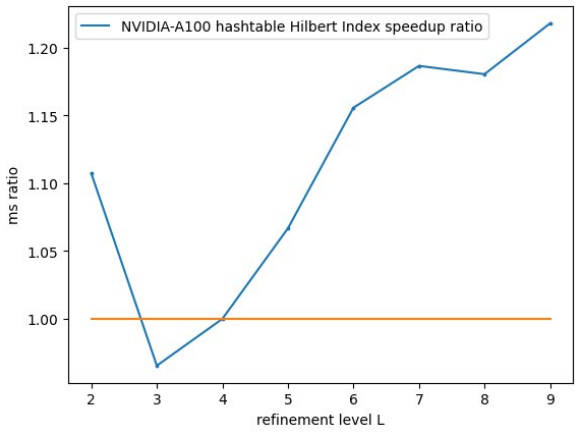

Figure 21. Gradient calculation performance speedup ratio on A100 with hashtable (Hilbert Index to row-major order)

cells. Currently the cells are stored in the exact Hilbert order. However, in a situation when cells are added and removed from the array, maintaining exact Hilbert order may be impractical. A possible approach in this case can be that we keep half of an array empty for resizing purposes. Then as needed, we remove cells and add new cells in the empty half of the array. We keep track of the current cell order and ideal Hilbert cell order. When there is too much difference in the orders or too many gaps, we can reorder the array from scratch. In this context we would need a metric to assess how much the array order differs from desired. Possibly around the midline between the ideal order and a completely wrong order we would rewrite the array. The exact point at which to rewrite the array can be optimized by the total speed of the algorithm. The metric used to assess the deviation from the ideal order can be the average difference between the pairwise distances of cell indices in the current order and those in the ideal order:

$$\frac{2}{N(N-1)} \sum_{i,j} ||i_{curr} - j_{curr}| - |i_{hilbert} - j_{hilbert}||$$

In this future implementation we can benefit from `cuco::dynamic_map` in place of the `static_map` we are using now. `static_map` is notably fast on GPU when the hashtable size is predictable, because it is well optimized for a constant size. On the other side, resizing of `dynamic_map` may lead to memory fragmentation, in other words scattered memory allocation. This leads to slow performance. Additionally, there is an overhead for the resizing feature, which may lead to slower performance for small L. However, if resizing is required, the use of `dynamic_map` is unavoidable and beneficial for higher L.

6 CONCLUSION

My work builds on initial contributions from the COS 521 group project Andalman, Z. et al. (2023), but results in a working implementation of the hashtable-based AMR on GPU as part of my independent research, including results I achieved in the COS 597E project.

In summary, my main contribution is a working implementation of the hashtable for a toy problem of AMR on GPU. Our results confirm that the hashtable approach maintains competitive performance with the matrix implementation in fully refined grids. More importantly, it is expected to significantly outperform the matrix approach in non-uniformly refined grids.

The matrix implementation must store all possible cells (i, j, k, L) in the grid, even if they do not exist. In contrast, the hashtable efficiently avoids storing most nonexistent cells. As a result, performance time scales linearly with the number of stored cells. This suggests that the hashtable implementation is suited well for real-world applications with non-uniform refinement such as astrophysical shock waves, where the small cells are concentrated in the areas of interest.

Moreover, a speedup of ~2.3× for $L = 9$ was found from running a hashtable implementation on Grace Hopper Superchip GH200

compared to running on a regular GPU A100. This speedup of the gradient calculation is more significant compared to speedup of ~14.7× for hashtable creation. In a complete AMR simulation, we are expected to create the hashtable only once, and then perform continuous hydrodynamics calculations, along with continuous refinement and unrefinement of cells in the areas of interest. Nevertheless, the AMR GPU Future Work subsection discusses potential resizing of the hashtable due to cell refinement and unrefinement. This would require recreating the hashtable from scratch, utilizing the ~14.7× speedup.

Our analysis leads to the following key results.

The hashtable-based AMR implementation on GPU does not cause a significant slowdown when the grid is fully refined. In a practically relevant case of a non-uniform distribution of small cells, the hydrodynamics calculation time is expected to scale linearly with the total number of cells.

Using the Grace Hopper GH200 provides additional speedup in gradient calculations and hashtable creation compared to the A100 GPU.

As a result, it is practical to incorporate a GPU-based hashtable into an AMR implementation with a complete hydrodynamics simulation such as mini-ramses[15], and to run it on a cutting-edge superchip such as Grace Hopper GH200.

REFERENCES

Amazon 2024, AWS and NVIDIA Extend Collaboration to Advance Generative AI Innovation, https://press.aboutamazon.com/2024/3/aws-and-nvidia-extend-collaboration-to-advance-generative-ai-innovation

Andalman, Z. Nataliia Khotiaintseva Roma Bhattacharjee 2023, Technical report, GPU-Acceleration of Adaptive Mesh Refinement Algorithms. Princeton University Computer Science Department

Biernacki P., Teyssier R., Bleuler A., 2017, Monthly Notices of the Royal Astronomical Society, 469, 295

Commerçon B., Teyssier R., Audit E., Hennebelle P., Chabrier G., 2011, A&A, 529, A35

Godunov S. K., 1959, Mat. Sb., Nov. Ser., 47, 271

Juenger D., Iskos N., Wang Y., Hemstad J., Hundt C., Sakharnykh N., 2023, Maximizing Performance with Massively Parallel Hash Maps on GPUs

Khotiaintseva, N. 2024, Technical report, Warp Divergence Linter Program and Grace Hopper GPU Superchip run for CUDA C++ Adaptive Mesh Refinement Code. Princeton University Computer Science Department

Menshov I., Pavlukhin P., 2020, in Journal of Physics Conference Series. p. 012017, doi:10.1088/1742-6596/1640/1/012017

NVIDIA 2015, Divergent Branch, https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/divergentbranch.htm

NVIDIA 2021, Nvidia A100 Tensor Core GPU Datasheet, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

NVIDIA 2023, AWS and NVIDIA Announce Strategic Collaboration to Offer New Supercomputing Infrastructure, Software and Services for Generative AI, https://press.aboutamazon.com/2024/3/aws-and-nvidia-extend-collaboration-to-advance-generative-ai-inno

NVIDIA 2024a, CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

NVIDIA 2024d, CUDA-GDB, https://developer.nvidia.com/cuda-gdb

NVIDIA 2024f, CUDA-MEMCHECK, https://developer.nvidia.com/cuda-memcheck

NVIDIA 2024g, NVIDIA Compute Sanitizer, https://developer.nvidia.com/compute-sanitizer

NVIDIA 2024h, Nvidia Grace Hopper superchip data sheet, https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip?ncid=no-ncid

NVIDIA 2024i, Nvidia Grace Hopper superchip data sheet, https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip

NVIDIA 2024b, Nvidia Nsight Compute, https://developer.nvidia.com/nsight-compute

NVIDIA 2024e, NVIDIA Nsight Graphics, https://developer.nvidia.com/nsight-graphics

NVIDIA 2024c, NVIDIA Nsight Systems, https://developer.nvidia.com/nsight-systems

Schive H.-Y., ZuHone J. A., Goldbaum N. J., Turk M. J., Gaspari M., Cheng C.-Y., 2018, MNRAS, 481, 4815

[15] https://bitbucket.org/rteyssie/mini-ramses

Skilling J., 2004, in Erickson G. J., Zhai Y., eds, American Institute of Physics Conference Series Vol. 707, Bayesian Inference and Maximum Entropy Methods in Science and Engineering. pp 381–387, doi:10.1063/1.1751381

Teyssier R., 2002, A&A, 385, 337

This paper has been typeset from a TEX/LATEX file prepared by the author.