SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Guided Research in Computational Science and Engineering

Optimizing GPU Offloading with CUDA for a Patch-based Hyperbolic Finite Volume Solver in ExaHyPE

Ahmed Fouad

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Guided Research in Computational Science and Engineering

Optimizing GPU Offloading with CUDA for a Patch-based Hyperbolic Finite Volume Solver in ExaHyPE

Author:	Ahmed Fouad
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Mario Wille, M.Sc.
Submission Date:	22.10.2023

I confirm that this guided research in computational science and engineering is my own work and I have documented all sources and material used.

Munich, 22.10.2023 Ahmed Fouad

Acknowledgments

I express my deepest gratitude to Mario Will for his unwavering support and guidance throughout my research in optimizing the finite volumes Rusanov solver in CUDA. His unique blend of professionalism and flexibility has been instrumental in shaping both my research and my growth as a scholar. His dedication to my work and his constant availability have provided a strong foundation upon which I built my research. I am truly fortunate to have had him as my advisor and am immensely thankful for the insights, encouragement, and expertise he has generously provided.

Abstract

The present study focuses on optimizing GPU offloading techniques within the Peano computational framework for adaptive mesh refinement (AMR) in Euler simulations using the ExaHyPE 2 engine. We first investigate the ExaHyPE 2 finite volumes Rusanov kernels to evolve the system of Euler equations using the CUDA framework. Then, we systematically identify and mitigate performance bottlenecks in the kernels' offloading process, yielding measurable overall improvements in the GPU computational efficiency.

Contents

1 Introduction

In this section, we describe the primary concepts of the Rusanov solver, highlight their roles, and discuss their representation in ExaHyPE 2.

1.1 Peano and ExaHyPE 2

Peano is a framework for solvers operating on dynamically adaptive Cartesian meshes. It is a base framework for further extensions, such as the ExaHyPE 2 engine and various toolboxes, while Peano is only about mesh management, data storage, distribution, and mesh traversal.

The Peano framework allows different solver engines to operate on dynamic adaptive Cartesian grids. Although it serves as the foundation for various toolboxes tailored to distinct applications, its primary focus remains on mesh management, data retention, distribution, and mesh navigation. ExaHyPE 2 is one of the engines that utilizes the Peano framework mesh processing capabilities. Peano and ExaHyPE 2 operate in synergy to enable dynamic adaptivity on Cartesian grids using spacetree data structures as shown in Figure 1.1.

ExaHyPE 2 (An Exascale Hyperbolic PDE Engine) is a specialized computational platform that simulates hyperbolic Partial Differential Equations (PDE)s (cf. [11]). With its orientation toward exascale computing architectures capable of performing at least one exaFLOP, the platform is a High-Performance Computing (HPC) nexus for various scientific applications. The engine capitalizes on advanced numerical techniques, including Finite-Volume and Discontinuous Galerkin methods, to address hyperbolic PDEs. These methods are executed on dynamically adaptive computational meshes, enabling localized, high-fidelity solutions that optimize resource allocation.

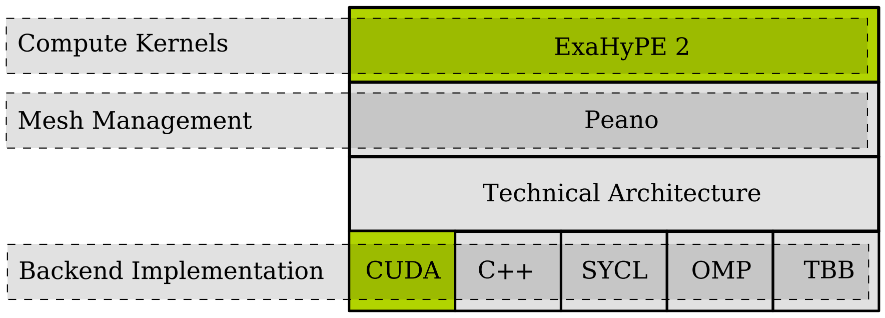


Figure 1.1: Schematic representation of ExaHyPE 2, Peano, and their CUDA integration.

This study focuses on optimizing the GPU offloading processes within ExaHyPE 2 by employing NVIDIA's profiling toolbox. The study aims to identify and alleviate computational bottlenecks in the Rusanov finite volumes solver.

1.2 Rusanov Solver

1.2.1 Spatial Fluxes Conservation

Rusanov's method is a numerical scheme for solving hyperbolic PDEs. In the finite volume methods context, the Rusanov method is often implemented as a solver kernel to approximate fluxes across cell interfaces in a discretized computational domain. It conserves fluxes by averaging them from adjacent cells and correcting them with a dissipative term proportional to the difference in states of the adjacent cells. This helps capture the shocks and discontinuities in hyperbolic conservation laws (cf. [12]).

Mathematically, given the local state $\mathbf{Q}$ with three-dimensional state components:

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix} \tag{1.1}$$

where ρ is the density, u and v, w are the x-, y-, and z-components of velocity, and E is the total energy per unit volume.

The Rusanov flux, for two local states Q_i and Q_j, is expressed as:

$$F_{\text{Rusanov}}(\mathbf{Q}_L, \mathbf{Q}_R) = \frac{1}{2}\left[\mathbf{F}(\mathbf{Q}_L) + \mathbf{F}(\mathbf{Q}_R)\right] - \frac{1}{2}S(\mathbf{Q}_R - \mathbf{Q}_L) \tag{1.2}$$

Here, $\mathbf{Q}_L$ and $\mathbf{Q}_R$ are the states at the left and right sides of the interface, and S is the wave speed, often approximated as the maximum eigenvalue λ of the Jacobian of

F. λ represents the maximum eigenvalue of the Jacobian of the flux function F and is expressed as:

$$\lambda = \max\left(|u \pm c|\right) \tag{1.3}$$

We can then find the speed of the wave c :

$$c = \sqrt{\gamma \frac{|p|}{|\rho|}} \tag{1.4}$$

where γ is the specific heat ratio and approximated by 1.4 in our simulation and the pressure term p is defined as:

$$p = (\gamma - 1)\left(E - \frac{\rho}{2}(u^2 + v^2 + w^2)\right) \tag{1.5}$$

Equation 1.2 is general and works for both 2D and 3D cases as long as the appropriate flux function $\mathbf{F}(\mathbf{Q})$ and the state vector $\mathbf{Q}$ are used for the respective dimensions.

For each cell interface i, $\mathbf{F}(\mathbf{Q_i})$ is the flux function. The flux vector $\mathbf{F}$ together with state vector $\mathbf{Q}$ can be used to conserve spatial fluxes across cell interfaces. Using conservation laws, we obtain:

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial \mathbf{F_x}(\mathbf{Q})}{\partial x} + \frac{\partial \mathbf{F_y}(\mathbf{Q})}{\partial y} + \frac{\partial \mathbf{F_z}(\mathbf{Q})}{\partial z} = 0 \tag{1.6}$$

By plugging in the full definition of $\mathbf{Q}$ from Equation 1.1, Equation 1.6 expands to:

$$\frac{\partial}{\partial t}\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ \rho u w \\ (E+p)u \end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ \rho v w \\ (E+p)v \end{bmatrix} + \frac{\partial}{\partial z}\begin{bmatrix} \rho w \\ \rho u w \\ \rho v w \\ \rho w^2 + p \\ (E+p)w \end{bmatrix} = 0 \tag{1.7}$$

The first term is the temporal conservation term. The pressure term p is added in the diagonal of the momentum components.

The remaining terms in Equation 1.7 represent the spatial conservation of ρu, $\rho u^2 + p$, ρuv, and $(E + p)u$ in the horizontal x-direction, ρv, ρuv, $\rho v^2 + p$, and $(E + p)v$ in the vertical y-direction, and analogous terms for the depth-directed z-flux.

1.2.2 Temporal Wave Speed Conservation

A numerical procedure for evolving the solution in time is usually used with Rusanov in computational fluid dynamics (CFD) applications. A time stepping scheme is often used to integrate the ordinary differential equations (ODEs) resulting from the PDEs' spatial discretization as we usually conserve fluxes spatially in higher dimensional space.

In the finite volume method context, after the spatial fluxes are computed, a time-stepping kernel can be invoked to evolve the cell-averaged solution from one level to the next.

To ensure stability, the maximum eigenvalue corresponding to the maximum wave speed in Equation 1.2 determines the local time step (or CFL condition). We guarantee temporal stability over multiple iterations in the simulation using the maximum eigenvalue λ.

Mathematically, a general form of an explicit time-stepping scheme for the state $\mathbf{Q}$ can be expressed as:

$$\mathbf{Q}^{n+1} = \mathbf{Q}^{n} - \Delta t \nabla \cdot \mathbf{F}(\mathbf{Q}^{n}) \tag{1.8}$$

Here, $\mathbf{Q}^{n}$ and $\mathbf{Q}^{n+1}$ are the cell-averaged states at time levels n and $n+1$, respectively,. Δt is the time step, and $\nabla \cdot \mathbf{F}$ represents the divergence of the flux vector, often approximated using fluxes computed by the Rusanov solver at cell interfaces in Equation 1.2.

For the conservation laws in three-dimensional space, Equation 1.8 can be adapted to:

$$\mathbf{Q}_{i}^{n+1} = \mathbf{Q}_{i}^{n} - \frac{\Delta t}{\Delta x}\left(\mathbf{F}_{i+\frac{1}{2}} - \mathbf{F}_{i-\frac{1}{2}}\right)$$

$$\mathbf{Q}_{i,j,k}^{n+1} = \mathbf{Q}_{i,j,k}^{n} - \frac{\Delta t}{\Delta x}\left(\mathbf{F}^{x}_{i+\frac{1}{2},j,k} - \mathbf{F}^{x}_{i-\frac{1}{2},j,k}\right) - \frac{\Delta t}{\Delta y}\left(\mathbf{F}^{y}_{i,j+\frac{1}{2},k} - \mathbf{F}^{y}_{i,j-\frac{1}{2},k}\right) - \frac{\Delta t}{\Delta z}\left(\mathbf{F}^{z}_{i,j,k+\frac{1}{2}} - \mathbf{F}^{z}_{i,j,k-\frac{1}{2}}\right) \tag{1.9}$$

Where each Rusanov flux component is calculated as shown in Equation 1.2.

Using the above numerical model, we implemented the computational kernels of the solver in ExaHyPE 2, which we later profiled, optimized, and tried to mitigate their bottlenecks.

1.3 CUDA

The Compute Unified Device Architecture (CUDA) is a software framework developed by NVIDIA to expand the capabilities of GPU acceleration (cf. [10]).

1.3.1 Execution Model

Streaming Multiprocessors

In NVIDIA GPUs, the architecture revolves around scalable *streaming multiprocessors* (SMs). Each SM handles the simultaneous execution of thread groups. Once assigned to an SM, a thread group *scheduled warps* remains there until completion. An SM encompasses processing cores, shared memory (L1 cache), registers, a load/store unit, and a threads-scheduling unit.

Warps

Threads in CUDA are the finest execution units in CUDA. They are grouped into *blocks*, which are further grouped into a *grid*. Within blocks, threads are further clustered into *warps*. A warp typically consists of 32 threads that execute the same instruction simultaneously, making it the fundamental unit of parallelism in CUDA (cf. Figure 1.2). When a warp encounters a particular instruction in the kernel, all 32 threads execute it in parallel on their respective data. If there's a branching instruction and not all threads take the same branch, then threads taking different paths are masked and executed serially. This can lead to what is called *warp divergence*. When all different paths are finished, the warp reconverges and continues executing in lockstep. Multiple warps are scheduled simultaneously via a unit called the *scheduler*, which belongs to the SM. Each scheduler maintains a pool of warps that it can issue instructions for. The kernel launch configuration limits the upper bound of warps in the pool *theoretical warps*. Every cycle, each scheduler checks the state of the allocated warps in the pool for *active warps*. Active warps that are not stalled are eligible for the next instruction execution and are known as *eligible warps*. From the set of eligible warps, the scheduler selects a single warp to issue one or more instructions; once a warp is issued an instruction via the *issue slot*, the warp becomes an *issued warp*. On cycles with no eligible warps, the issue slot is skipped, and no instruction is issued. Having many skipped issue slots indicates poor latency hiding and, thus, overall low bandwidth.

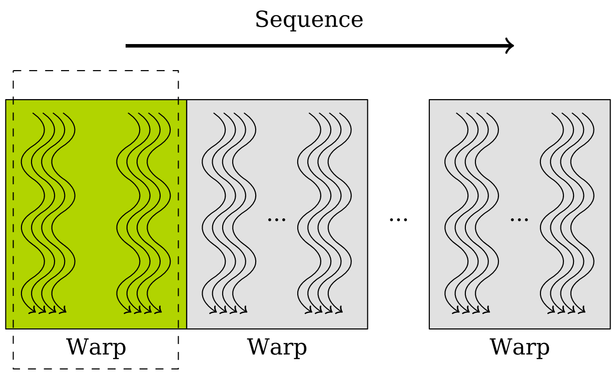


Figure 1.2: Representation of the execution policy in CUDA via warps, where each warp consists of 32 threads executing an instruction simultaneously.

Warp Divergence

Due to the Single Instruction, Multiple Thread (SIMT) architecture of NVIDIA GPUs (cf. [6]), all threads in a warp execute the same instruction on different data. This means that if two threads in the same warp need to execute different instructions due to a branching condition, the warp will serialize the different branches, leading to reduced performance. This serialization due to differing instruction paths within a warp is known as *warp divergence*.

Warp divergence can have a significant performance impact. If threads within a warp take different execution paths, the warp as a whole must execute each path serially, reducing the effective parallelism. This is because a warp can only execute one instruction at a time. Thus, frequent usage of branching `if-else` and nested logic streams will cause some threads in a warp to execute different paths, significantly reducing the warp's overall parallelism efficiency.

1.3.2 Memory Hierarchy

The memory hierarchy plays a pivotal role in influencing the performance and efficiency of GPU-accelerated applications. A generic CUDA memory hierarchy architecture is illustrated in Figure 1.3. A summary of the memory characteristics is listed in Table 1.1.

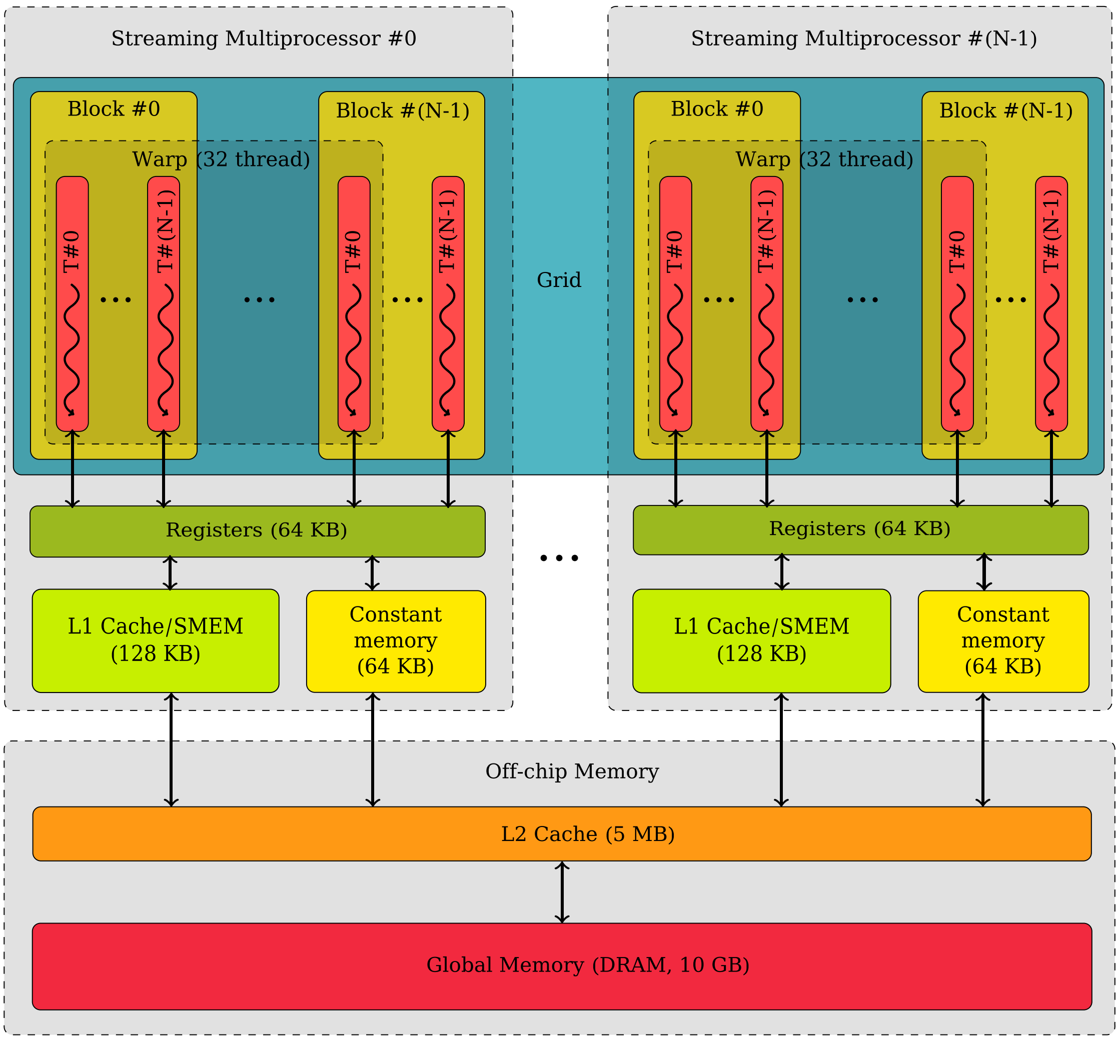


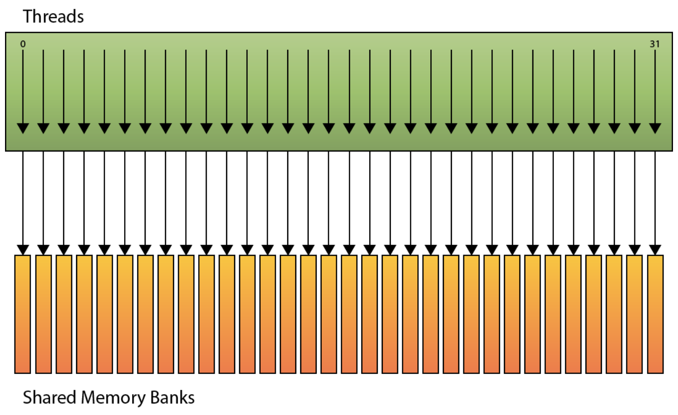
Figure 1.3: Memory hierarchy of the RTX 3080 CUDA-based GPU architecture.

Registers

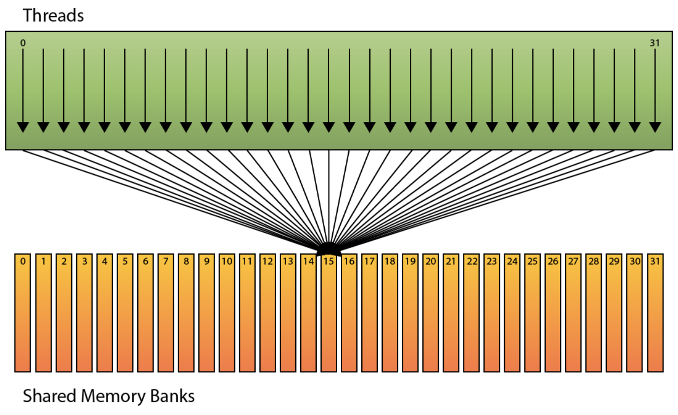
Registers in CUDA represent the fastest type of memory available and are located directly on the GPU's SMs. Each thread running on an SM has its own private set of registers.

Shared Memory

Shared memory (SMEM) is an on-chip memory type that allows threads within the same block to share data. It has much lower latency than global memory (GM) (cf. subsubsection 1.3.2) and is divided among thread blocks. SMEM is exclusive to a block of threads during their lifetime. Each block has its shared memory space that its threads can use collaboratively. As shown in Table 1.1, SMEM and L1 caches are configurable via CUDA API calls. Since they are both on the chip, SMEM can be considered a *software-controlled L1 cache*.



(a) The k^{th} thread accesses the k^{th} SMEM bank and no bank conflict occurs.



(b) No bank conflicts occur as long as all threads in the warp access the same word in the SMEM bank. This is known as *Broadcast*. All threads receive the same value simultaneously, broadcasted from that address.

Figure 1.4: Different situations where bank conflicts do not occur.

Despite having low latency, SMEM can suffer from bank conflicts that can degrade the bandwidth. Bank conflicts arise when multiple threads of a warp access different addresses that map to the same memory bank (cf. [9]). SMEM in NVIDIA GPUs is divided into multiple banks (usually 32 banks in most architectures), and each bank can service one request per cycle. Ideally, if all threads of a warp access a different bank, the access can be done in parallel, achieving maximum throughput (cf. Figure 1.4).

However, when two or more threads of a warp access different addresses within the same bank, a bank conflict occurs (cf. [2]). This causes serialization of the accesses, resulting in a performance penalty. The number of clock cycles it takes is proportional to the number of threads contending for that bank. Thus, it is best to make data accesses within the same thread span different addresses *i.e., with stride* in the SMEM to reduce such conflicts (cf. Figure 1.5). SMEM can be set to define the bank sizes in some CUDA architectures. In our setup, we left it to the default value of 4 Bytes per bank. It can be changed via `cudaDeviceSetSharedMemConfig(cudaSharedMemConfig)` to a maximum of 8 Bytes per bank.

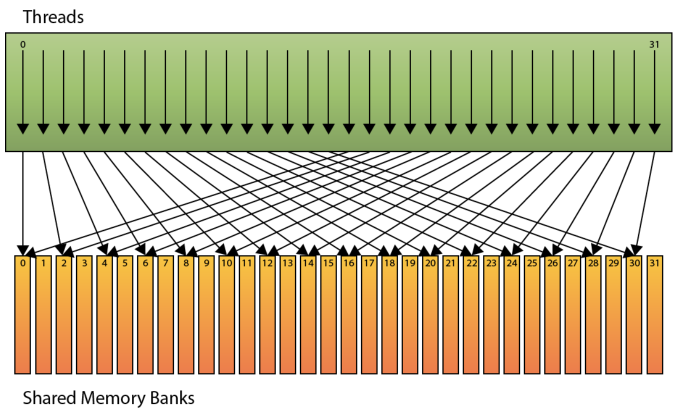


Figure 1.5: A 2-way bank conflict occurs when threads access different words in the same bank.

L1/L2 Cache

L1 caches are designed to bridge the gap between the processing cores and the GM. It provides low-latency access to recently used data, speeding up the data retrieval process for threads. However, due to their small sizes (typically in the range of 100 KBs), improper memory access patterns can cause frequent spilling into L2 caches and possibly to the GM when the data are too big (L2 can be up to 5 MB in the RTX 3080 GPU), rendering the two-level caching obsolete during the runtime and heavily reducing the throughput of the overall computation pipeline. Each SM has its own L1 cache. Hence, it is exclusive to threads running on the same SM. However, L2 caches belong to the entire set of SMs. It is shared among all SMs in a GPU.

Constant Memory

Constant memory is a small section of memory used to store read-only constants for the duration of a kernel execution. Access to constant memory is optimized to be faster than GM when all threads read the same location. All threads in all SMs can read from constant memory. It is globally accessible, though it is cached; repeated accesses to the same address are usually faster than GM accesses.

Local Memory

Local memory in CUDA is an off-chip memory used primarily for spilling registers when the register file overflows. Each thread has its private local memory. When a CUDA program declares an array as a local variable in the kernel and the size of the array cannot be determined at compile time, or if it is too large to fit into the available registers, it gets placed into local memory. Despite being local to a thread, this memory is not as fast as registers nor SMEM. Subsequent accesses to these spilled variables leverage the GPU's hierarchical memory structure. Initially, the GPU checks the L1 cache for the required data. If a miss occurs, the L2 cache is consulted. In the event of a cache miss in both L1 and L2, the data is then fetched from off-chip GM. The GPU's caching mechanism, sensitive to both spatial and temporal locality, can enhance the retrieval speed of frequently accessed local variables, reducing the need for slower GM fetches. It is important to note that local memory accesses are not cached in the L1 cache by default (but can be enabled on specific architectures).

Global Memory

GM resides in the device memory and provides a main DRAM large storage area accessible by all threads and the host (CPU). It has high latency and is not cached (although there are exceptions in recent architectures with L2 cache). GM is available to all threads regardless of their block or grid. It is the main off-chip memory of the GPU with considerably low bandwidth. In accelerated applications, we usually minimize memory requests from and to GM.

MEMORY	**SCOPE**	**SPEED**	**SIZE**	LOCATION
Registers	Per thread	Fastest	Smallest	On-chip
Shared	Per block	Fast	Small*	On-chip
L1 cache	Per SM	Fast	Small*	On-chip
L2 cache	All SMs	Faster than GM	Large	Off-chip
Constant	All threads and CPU	Slow, cached	Small	Off-chip
Local	Per thread	Slow, cached	Small	Off-chip
Global	All threads and CPU	Slowest, cached	Largest	Off-chip

Table 1.1: Overview of CUDA memory types and some general characteristics of each. *The SMEM and L1 cache are configurable in complementary manner. Together, they share the same on-chip memory storage. Either size can be set via CUDA APIs.

2 Methodology

2.1 Overview

In the computational implementation context, each Rusanov kernel calculates the numerical fluxes for each interface in the computational mesh. The kernel takes in the states $\mathbf{Q}_L$ and $\mathbf{Q}_R$ as input, computes the numerical flux F_{Rusanov}, and updates the states for the next time level.

For high performance, these kernels are often optimized to exploit the memory hierarchy of the hardware platform, in this case, the GPU. Special considerations may include coarsening the computational grid, optimizing the memory access patterns, reducing undesired memory transfers, and using advanced CUDA features for asynchronous operations to support overlapping the computation and memory migrations using CUDA *graphs* (cf. [3]) or *streams* (cf. [5]).

The computational complexity of each Rusanov kernel is generally $O(1)$ for each interface, leading to an overall complexity of $O(N)$ for N interfaces in the mesh. However, optimization techniques can influence the effective throughput.

As the basis of this study, we reviewed the results concluded in the GPU offloading managed by the CUDA Unified Memory Architecture (UMA) in [13]. That study has shown that after transitioning to the CUDA unified memory in place of manual copies, the benefits of concurrent GPU offloading are still negligible, even detrimental to the overall runtime. This approach delays the data movement cost until the data is needed, distributing bandwidth needs over time. However, this doesn't lead to any performance gains. Hence, we focus in our study on analyzing the bandwidth pipeline further using the CUDA profiling toolbox to gain deeper insights into the reasoning behind the bandwidth limitation.

2.1.1 Grid Traversal in Space and Time

The Rusanov solver and a temporal update scheme serve two different but complementary roles in the simulation pipeline of hyperbolic PDEs. A typical mesh traversal would be discretized in both *space* and *time* domains:

1. The Rusanov solver is used for spatial discretization, computing numerical fluxes at volume interfaces based on the governing hyperbolic PDEs.

2. The temporal update scheme is used for the temporal discretization, advancing the volume-averaged solution in time based on the computed numerical fluxes.

Thus, the time-stepping scheme often uses the output of the Rusanov solver to evolve the system's state over a timestep. The two are intrinsically linked: the Rusanov solver provides the spatially discretized form of the equations. At the same time, an update scheme method advances this discretized form in time (cf. Figure 2.1).

By understanding this symbiotic relationship (spatially and temporally), we gain more insights into the overall computational pipeline as it involves floating point operations (FLOPs) on different elements of the grid volume's state vectors, allowing for targeted optimizations, particularly in high-performance computing environments like GPUs.

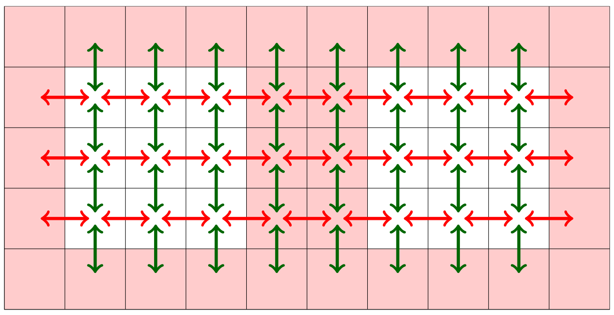


Figure 2.1: Exemplary computational grid for the 2D Euler equations with Rusanov fluxes.

The mathematical representation of a 2D grid (cf. Figure 2.1) is shown in Equation 1.7. The red and green arrows represent the spatial flux differentiation and maximum eigenvalue flow in the x- and y-directions, respectively. In this exemplary grid, the grid consists of 2 patches and 5×5 volumes per patch, counting in total to 50 volumes. The 5 volumes per patch axis consist of two main types of volumes: the white volumes counting to 3 per patch represent the *actual original grid* and red volumes counting $5 \times 5 - 3 \times 3 = 16$ per patch represent each patch's boundary for inter-patch communication and information flow. Those boundary volumes are called *halo or ghost volumes*. They are mainly used to store the latest grid state values on the boundary of each patch to allow neighboring patches to access the latest evolution of the communicating patch.

2.1.2 Original Implementation of the Solver

The local state vector for the grid **Q** in Equation 1.7 is represented as Q_{in} in our implementation. Q_{in} will always represent the input grid state in the previous timestep, and by using it, we calculate the system state evolution in the current timestep Q_{out}. By analyzing the computation kernels that take Q_{in} as input and contribute to calculating Q_{out} in the Rusanov solver, we could introduce tuned optimizations to the kernels' computations and memory access needs. The calculations access four physical entities per volume in Q_{in} that must be accessed to compute the terms presented in Equation 1.7 for every volume in every time step.

We start by listing those computations. This involves the following steps:

1. Calculate the temporal evolution (maximum eigenvalue).
2. Calculate the spatial fluxes.
3. Update the patch boundary for inter-patch-data-flow for the grid-level solution.
4. Reduce the maximum eigenvalue within the patch.
5. Update the output grid with the new calculated grid state.

We provide the pseudo-code for the first two stages because they involve the highest number of FLOPs and data accesses of *read/write*. The maximum eigenvalue and spatial fluxes calculations are shown in 0 and 0, respectively.

Algorithm 1 Eigenvalue computation algorithm for Euler equations

```
1: procedure computeMaxEigenvalue(Q, normal)
2:     $\gamma \leftarrow 1.4$
3:     irho $\leftarrow \frac{1}{|Q[0]|}$
4:     $p \leftarrow (\gamma - 1) \times \left(Q[3] - 0.5 \times \text{irho} \times (Q[1]^2 + Q[2]^2)\right)$
5:     $c \leftarrow \sqrt{\gamma \times |p| \times \text{irho}}$
6:     $u_n \leftarrow Q[normal + 1] \times \text{irho}$          ▷ Dimension specific update
7:     return $\max(|u_n + c|, |u_n - c|)$          ▷ Return Maximum Eigenvalue
8: end procedure
```

Algorithm 2 Flux computation for the 2D Euler equations

```
 1: procedure computeFlux(Q, normal, F)
 2:     $\gamma \leftarrow 1.4$
 3:     irho $\leftarrow \frac{1}{|Q[0]|}$
 4:     $p \leftarrow (\gamma - 1) \times \left(Q[3] - 0.5 \times \text{irho} \times (Q[1]^2 + Q[2]^2)\right)$
 5:     coeff $\leftarrow$ irho $\times Q[\text{normal} + 1]$
 6:     $F[0] \leftarrow \text{coeff} \times Q[0]$
 7:     $F[1] \leftarrow \text{coeff} \times Q[1]$
 8:     $F[2] \leftarrow \text{coeff} \times Q[2]$
 9:     $F[\text{normal} + 1] \leftarrow F[\text{normal} + 1] + p$
10:     $F[3] \leftarrow \text{coeff} \times Q[3] + \text{coeff} \times p$
11: end procedure
```

Overall, we can see that memory accesses are higher than the performed computations. Combining this with the fact that every physical term in Equation 1.7 is a state vector, memory alignment becomes a significant factor when accessing its elements non-uniformly in determining the efficiency of memory transfers to process the content of every volume as explained in subsection 1.3.2.

Several approaches are listed to address the limited bandwidth throughput:

1. Optimizing memory hierarchy.
2. Restructuring data access patterns.
3. Minimizing data transfers and use CUDA SMEM if available.
4. Overlapping independent data transfers and solver operations using CUDA streams.

To adopt the above strategies, we first need to analyze all the kernels that the CUDA Rusanov solver offloads to the GPU to solve the Equation 1.7 and update the grid in each time step. We will analyze in the next section the main compute and update kernels of the maximum eigenvalue and spatial fluxes.

2.2 Kernel Analysis

In this section, we present a deeper analysis of the kernels used by the solver. We offloaded those kernels into the Scientific Computing in Computer Science (SCCS) GPU cluster using the `x-wing0` node, which is equipped with NVIDIA RTX 3080 GPUs. Since our study focuses on CUDA optimization of the Rusanov kernels, we used a single GPU from the node. There, we also performed the benchmarks and obtained our results. This GPU has the following specifications:

Hardware Characteristic	Value
Global Memory (Avg. available)	10 GB
L1 Cache	128 KB (per SM)
L2 Cache	5 MB
SMEM	48 KB
FP64 (double)	465 GFLOPs
Memory Bandwidth	760.0 GB/s

Table 2.1: Hardware specifications of our test bench that consists of a single RTX 3080 GPU.

Each kernel in Table 2.2 represents a stage in the Rusanov solver except `rv-evolve`, a combination of those kernels to represent the solver as a whole in our benchmarking. From hereby on, we use the kernel labels introduced in Table 2.2 to distinguish between the stages in our base implementation of the Rusanov solver.

We start by copying the input grid into the solution grid as a basis for the current time step in `copy-sol`. Then, we calculate both maximum eigenvalue and spatial fluxes according to the Equation 1.7 in `comp-eigv` and `comp-flux`, respectively. During the calculations, we store intermediate values that we need to access via different kernels in temporary arrays of different sizes. The arrays are accessed every iteration from a GPU manager we implemented to avoid frequent allocations and de-allocations of those temporary arrays. Then, in the `update-sol` kernels, we update the mesh and utilize those temporary arrays. This sums up the stages that must be executed every time step by default. For profiling and comparison purposes, we also listed the performance of the entire Rusanov solver. This is represented in the `evolve` kernel, which contains all the aforementioned kernels mentioned here to show the overall performance.

Kernel Name	Kernel ID
Copy Q_in to Q_out	copy-sol
Compute Maximum Eigenvalue	comp-eigv
Compute Flux	comp-flux
Update Q_out	update-sol
Reduce Maximum Eigenvalue	reduce-eigv
Evolve (All previous kernels presenting the entire solver)	rv-evolve

Table 2.2: List of our finite volumes Rusanov CUDA kernels with their respective labels.

Computational Intensity

Our study focuses on the problem configurations presented in Table 2.3. The volumes are the total number of the grid volumes *including the interior volumes and the exterior* halo or ghost *volumes in the patches*. With the halo volume number mentioned between parenthesis per patch in each direction. For instance, a $30 \times 30 = 900$ interior volumes only mesh would expand to with halo to $32 \times 32 = 1024$):

Configuration Parameter	Value
Number of Patches	32,768, 16,384, 8,192
Number of Volumes Per Patch Axis	30(+2), 14(+2), 6(+2)

Table 2.3: Configurations of the test bench. The number of patches and volumes (We add 2 volumes per axis for halo per patch between parenthesis) are listed, and the results will list the Cartesian product of them.

We have chosen these configurations to ensure that we fully utilize the GPU warps as explained in subsection 1.3.1. 2 Gigabytes (GB) of data is enough to populate the cache levels (cf. Table 2.1) and proportionally sections of the GM as presented earlier in subsection 1.3.2. This forces the kernels to access data from GM more often to signify the memory access pattern's impact on the bandwidth. At the same time, we wanted to see the impact on the CUDA block level (size of the volumes per patch). In our calculations, we considered the halo volumes as a part of the input and processed grid by some kernels that need them, others have the halo volumes in their calculations excluded since no memory transfers *read/write* were done on the halo.

The computation throughput calculation is done by counting each FLOPs in each kernel, this will help us later when we calculate their arithmetic intensities. The memory throughput also known as the *Effective Bandwidth* in [2] is calculated using the following formula:

$$\text{Effective Bandwidth (GB/s)} = \frac{(B_{read} + B_{write}) \times 10^{-9}}{T} \tag{2.1}$$

We can count the FLOPs in the `cuda-fused` kernel *similarly for the rest, excluding or including the intermediate arrays that each kernel uses*, then count the memory bytes accessed by the kernel to compute its arithmetic intensity. We compute the total number of bytes generally according to the Equation 2.6:[1]:

$$\text{Input Grid Size (Bytes)} = P \times (V + Halo)^2 \times U \times 8 \text{ Bytes} \tag{2.2}$$

$$\text{Output Grid Size (Bytes)} = P \times V^2 \times U \times 8 \text{ Bytes} \tag{2.3}$$

$$\text{dt}(Bytes) = P \times \text{Double Precision Size} \tag{2.4}$$

$$\text{Cell Sizes}(Bytes) = P \times \text{Double Precision Size} * \text{Dimensions Size} \tag{2.5}$$

$$\text{Total Bytes} = \text{Input Grid Size} + \text{Output Grid Size} + \text{dt} + \text{Cell Sizes} \tag{2.6}$$

Where P, V, and U are patches, volumes/patch axis, and number of unknowns per volume, respectively.

Table 2.4 lists the collected information for each kernel for a single configuration of 32K patches and 30 interior volumes *32 with halo* per patch axis as follows:

Kernel ID	FLOPs	Global Memory Transfers (GB)	Arithmetic Intensity (AI)
copy-sol	0	2.02	0
comp-eigv	40	1.61	2.48×10^{-08}
comp-flux	34	3.22	1.06×10^{-08}
update-sol	50	4.70	1.06×10^{-08}
reduce-eigv	80	0.94	8.47×10^{-08}
rv-evolve	204	4.70	4.34×10^{-08}

Table 2.4: Arithmetic intensities of the kernels for the 32k patches and 30 volumes per patch axis.

[1]We consider double precision in our study.

We obtained the following results on our test setup executing the kernels in Table 2.2 for 100 time steps.

From the above information, we calculated and listed the Arithmetic Intensity as follows:

$$AI = \frac{FLOPs}{Bytes} \tag{2.7}$$

We can see that they have very low arithmetic intensity due to the constant movement of huge data arrays and relatively few FLOPs executions on them. Details of the FLOPs count can be found in the code on GitLab ExaHyPE 2.

In our offloading mechanisms, we mapped a single patch to a single CUDA block and a single volume to a single CUDA thread. Within each block, we could utilize higher bandwidth sections on the chip like CUDA SMEM. By testing $16 \times 16 = 256$ threads per block, we are testing a different warp occupancy since occupancy of warps in CUDA is a critical metric to ensure we have utilized all theoretically available threads per warp (problem dependent). Therefore, we included $32 \times 32 = 1024$ threads/block and a portion of that at 256.

Bandwidth Analysis

We ran the aforementioned kernels (cf. Table 2.2) and measured their bandwidths. As a baseline of the actually reachable peak bandwidth *because overhead can occur during the simulation run on the cluster*, we calculate the peak bandwidth via a simple copy kernel that copies all the content of one array to another that is located on GM. The measured peak bandwidth reported on the cluster was $681 GB/s$, which is relatively lower than the HW theoretical peak bandwidth. This might be due to interconnect overhead or simulation process execution rules set by the GPU cluster. To validate our theory, we executed multiple benchmarking kernels to measure bandwidth (inside and outside Peano). The result is that we have constantly obtained a peak of $681 GB/s$ using our reference copy kernel *not the Rusanov copy kernel* with different launch configurations and the CUDA samples bandwidth evaluation kernels (cf. [4]). The reported peak bandwidth the `CUDA bandwidthTest` kernel reached is $672 GB$. Thus, from now on, we will use the $681 GB/s$ as the peak bandwidth reference our kernels could target. Moreover, using the same reference copy kernels on our local machines, we reported a peak performance that matches the theoretical peak performance. In our local machine, we have a single RTX 3070 Mobile GPU with a reported peak of $445 GB/s$, which is very close to the theoretical peak of $448 GB/s$.

As can be seen in Figure 2.2, only a single kernel is getting close to the peak bandwidth possible in our system with 681 GB/s peak bandwidth, that is the `copy-sol` kernel at 581GB/s. Since it merely moves data one after another in constant stride-controlled accessed data from one array to another. This allows the GPU to use more elements of the previously requested sectors as explained in the memory access patterns in subsection 1.3.2. By calculating the efficiency of the bandwidth on the CUDA thread level of this kernel, we get a more concrete performance metric that we can use as a baseline in our tests.

By analyzing the results, we can see that the compute kernels `comp-eigv` and `comp-flux`are less efficient. This indicates an improper memory access pattern. Because our computation peak is at 450 GFLOPs (cf. Table 2.1) [2], and we perform considerably lower FLOPs than that, this underscores the fact that our arithmetic intensity is relatively very low to consider computation throughput in our analysis, as we will further elaborate in the rest of our analysis.

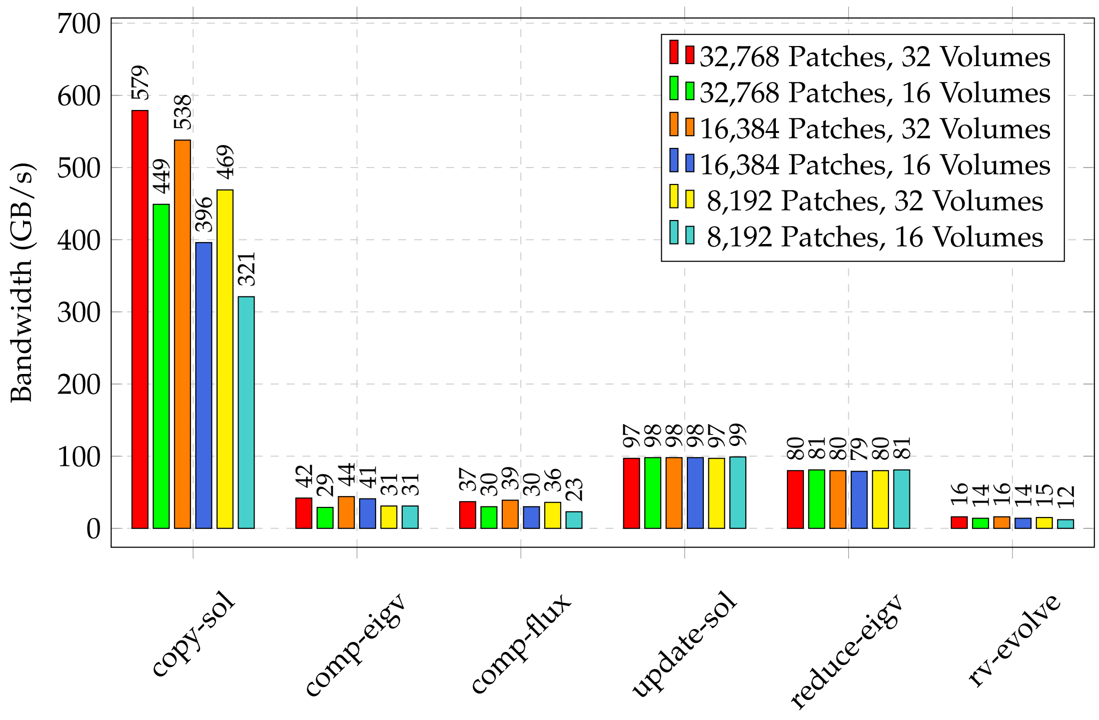


Figure 2.2: Bandwidth of all kernels of the original CUDA Rusanov solver. Each kernel is launched separately and timed with CUDA events to capture its execution time. Since different kernels handle different data, there are different sizes of *read and write* data transfers for each kernel.

[2]We consider only double precision in our study.

We can calculate the efficiency of the kernels on the solver level (entire kernel) or the CUDA thread level (representing a volume). For the solver level, we calculate the overall kernel efficiency *on CUDA grid level* as a percentage of the peak bandwidth using the following formula:

$$\text{Kernel Efficiency (GB/s)} = \frac{BW_{kernel}\ \text{GB/s} \times 100}{681\ \text{GB/s}} \tag{2.8}$$

This will give us the percentage of each kernel's overall efficiency. For the `copy-sol`, we obtained the highest score at 85% while the rest of the kernels scored in the average of 7.5%, indicating that we are facing a low memory throughput in compute and update kernels.

On the CUDA thread level, we can see the kernel efficiencies in Figure 2.3. The closest to peak, as explained earlier, is the `copy-sol` kernel at around 20%. We showed that the possible peak bandwidth in our experiments was $681GB/s$. Thus, we will use the 20% as a reference efficiency for the other kernel's threads. Some kernels work well separately, as in `comp-eigv` and `reduce-eigv`, which both scored higher than the overall `rv-evolve` reaching 1.2% and 5.8%, respectively. Nevertheless, the data dependencies between the computation, update, and reduction stages will persist, and thus, no parallelization could be employed between kernels such as CUDA streams or CUDA graph nodes. The other information we can infer from the results is that with an increasing number of blocks in the grid *patches*, allocating a single patch to a CUDA block is heavily impacted in computations. During computations, the vector is accessed multiple times from different threads to propagate the state information of the volumes across the neighbors. Improving the access by utilizing SMEM has somehow introduced stability in our results, which we will present in the next chapter.

To find out how efficient our threads are w.r.t. computation and memory transfers, we calculate the volume bandwidth efficiency as follows:

$$\text{Volume Efficiency} = \left(\frac{t}{N}\right) \div \left(\frac{t_{loc}}{P \times V}\right) \times 100 \tag{2.9}$$

Where t is the minimum possible time that was chosen as reference to the data-copying kernels illustrated in Figure 2.2, t_{loc} is the runtime duration of the CUDA kernel, N is a very large number that fills up all high bandwidth caches and memory and forces the kernel to reach out to GM, P is the number of patches and V is the number of volumes.

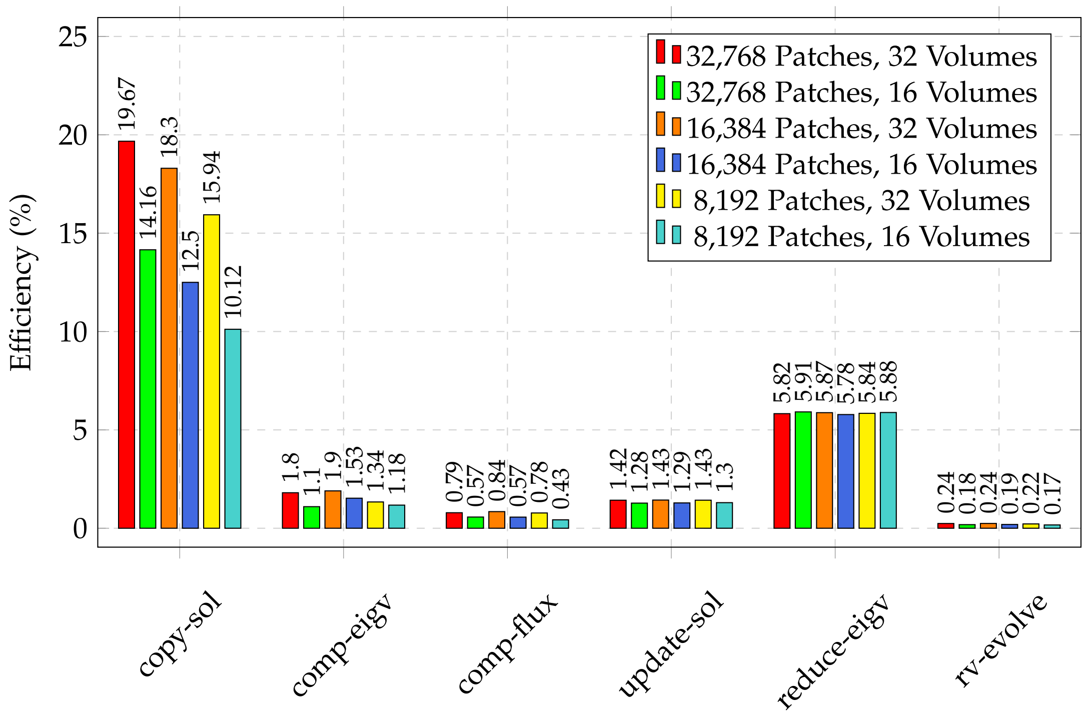


Figure 2.3: Efficiency on a CUDA thread level of all kernels of the original CUDA Rusanov solver as a percentage of the peak bandwidth.

After gaining insight into how extremely memory-bound our kernels are, we utilized NVIDIA's Nsight Compute detailed analysis section to determine how to improve it.

Nsight Compute [7] is a kernel-level profiler offering detailed performance metrics and insight into the performance of individual kernels. We analyzed the processing pipeline of the Rusanove solver to identify possible parallelizable paths. We profiled all kernels separately. Nsight Compute helps us optimize the data handled by each kernel. This gives us a finer granularity of problems that impact the bandwidth. We collected the most important findings that impact the memory bandwidth in Figure 2.4:

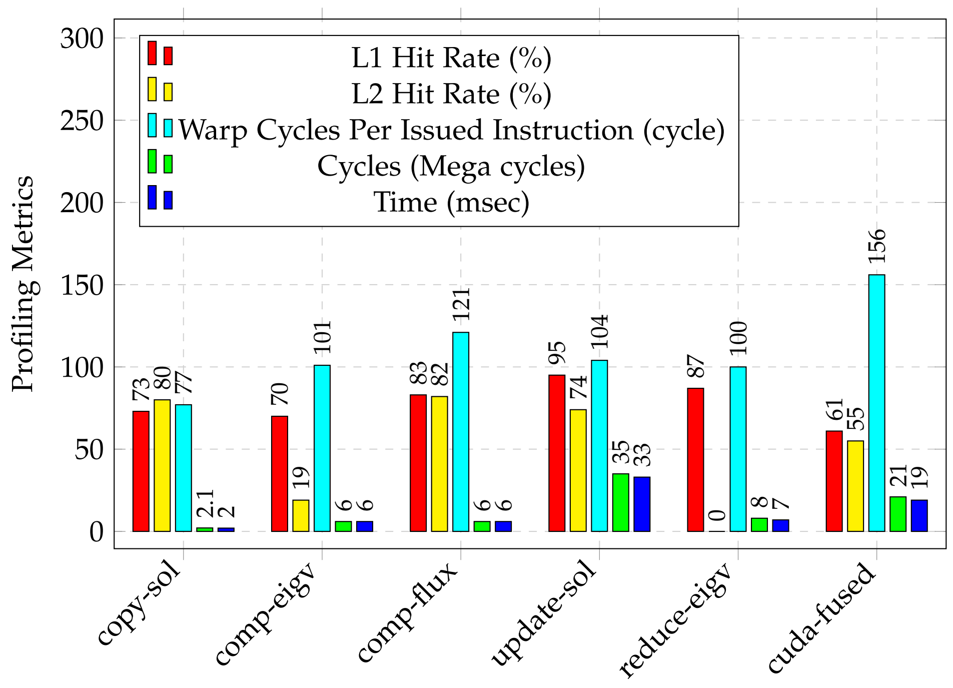


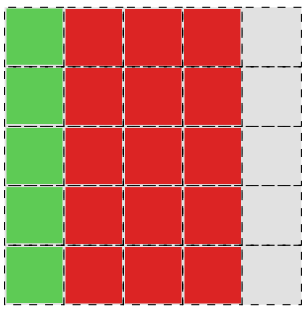
Figure 2.4: Cache hits, warp cycles, execution time reported by Nsight Compute.

On average, the kernels' warps spend around 109 cycles being stalled, waiting for an MIO (memory input/output) instruction in almost all kernels. Ideally, we are looking for less number of cycles between warps being active. The scheduler in RTX 3080 is capable of issuing one instruction per second. So by merging multiple kernels and reducing the access to GM, we are targeting less memory transfer and delegating that to SMEM as memory access there doesn't have to be fully coalesced for optimal loading. SMEM has low latency.

Memory Access Patterns

In addition to the above, Nsight compute reported excessive uncoalesced global memory accesses to the L2 in the `cuda-original` kernels. As explained in subsection 1.3.2, memory access patterns can cause significant degradation in performance if requests are not aligned well, which can cause cache thrashing or suboptimal usage of already fetched data by using only a portion of them. This is called *uncoalesced memory access*.

Uncoalesced memory access in the context of GPUs refers to threads within the same warp access non-contiguous memory addresses. This leads to inefficient memory access patterns, causing the memory subsystem to make more transactions to fetch the requested set of data elements than needed, as shown in Figure 2.5a with the red cells representing data that were fetched but not requested. Ideally, in a memory transfer, all fetched data into the cache line should also be processed and not discarded, as shown in Figure 2.5b.


(a) 5 different memory transfers initiated to load the target 5 elements of the grid, increasing the chances of frequent cache evictions.

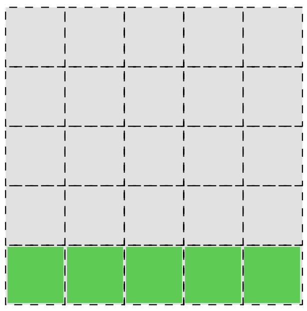
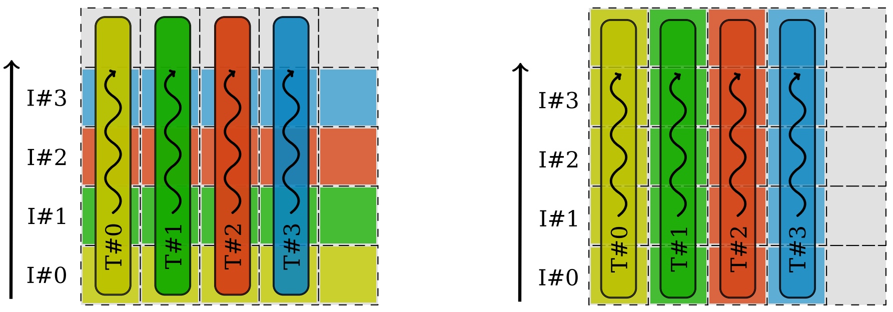

(b) 2 memory transfers are enough to fetch the target 5 data elements in a *coalesced* manner, increasing the spatial and temporal locality.

Figure 2.5: Memory access patterns with coalesced and uncoalesced accesses. Assuming 4 elements are fetched per request on a cache line level(if a cache line is 32 B and our data are double precision (8 B), 1 memory transfer fetches 4 elements), green elements represent the target elements to read, and red elements represent unused fetched data.

Since a warp consists of 32 threads, it is paramount to ensure that our warps adhere to coalesced access pattern. By applying the concept explained in Figure 2.5 to the CUDA warp, we obtain the memory access patterns shown in Figure 2.6. In Figure 2.6a, *I#0-3* represent instructions to be executed in the kernel by the warp threads one after another, and every thread executes the same instruction that was issued to the warp by the scheduler (cf. subsection 1.3.1). In the same instruction, if every thread loads its data separately (cf. Figure 2.6a), more memory transactions are performed. On the other hand, if *T#0* loads the first element and *T#1-3* access the subsequent elements already loaded by the first thread in the warp, we avoid that overhead.



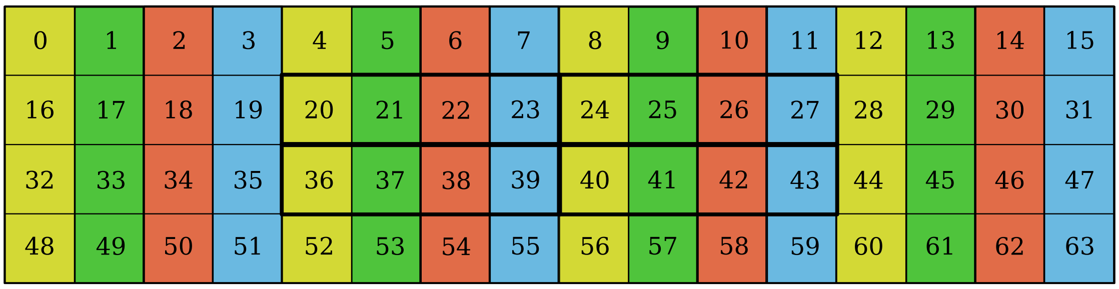
(a) Threads in a warp load data in an uncoalesced manner. Every thread accesses a data element not fetched with the data requested by other threads, causing pressure on the L1/L2 caches.

(b) Threads in a warp load data in a coalesced manner. The first thread *T#0* loads the first element of the row, and in the same memory transfer, the rest of the threads use the fetched data.

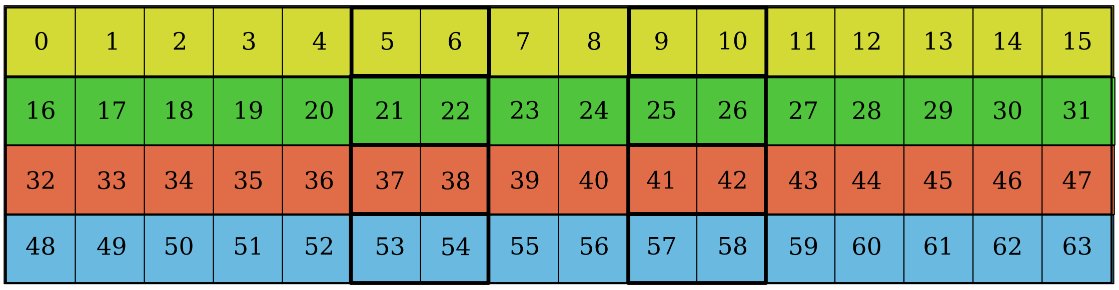
Figure 2.6: Coalesced and uncoalesced memory accesses in a warp.

Taking that into consideration while optimizing the Rusanov solver, we show in Figure 2.7 the aforementioned access patterns. Each color represents a state variable inside Q_{in}. This array contains 4 double precision elements for each volume, as shown. In our implementation, we counted for both approaches, picking the second one shown in Figure 2.7b, as it yielded (with a small margin) the best results.

When threads within a warp access consecutive memory locations, the hardware can fetch the data in a single, large memory transaction, which is much more efficient. This is termed coalesced memory access. In contrast, uncoalesced memory access patterns can significantly reduce the effective memory bandwidth, leading to performance degradation. In CUDA, memory is accessed in chunks known as *sectors*. Coalesced memory accesses are realized, and the threads in a warp use each sector fetched from GM efficiently. However, when we have uncoalesced accesses, the warp does not use all data from a fetched sector, leading to inefficiencies. In this context, the term **Excessive sectors** refers to the number of sectors fetched from GM that were not strictly necessary due to these uncoalesced accesses. If memory access were perfectly coalesced, fewer sectors would need to be fetched.



(a) Array of structure layouts that cause spatial locality for the running thread but force other threads in the same warp to access their elements in the grid with a stride in the interior volumes.



(b) Structure of arrays allows threads in the same warp to access adjacent elements without a stride in the interior volumes.

Figure 2.7: Exemplary grids of $(2+2)^2 = 16$ volumes. Each volume contains 4 values; in total, the renders $16 \times 4 = 64$ double precision elements to be accessed in the entire mesh, including both interior and halo volumes. Comparison of two main layouts of grid data with each volume in the grid containing four state values that will be processed. A thick black border surrounds interior volumes. The rest are the halo volumes.

Reported by Nsight Compute for the update kernels: 273×10^6 *excessive sectors* means that this many sectors were fetched in addition to what would have been fetched if all memory accesses were perfectly coalesced. *73% of the total* 273×10^6 *sectors* indicates that a significant portion (73%) of all sectors fetched were unnecessary or *excessive* due to uncoalesced memory access patterns.

This indicates a considerable inefficiency in the memory access pattern of the kernel, which could be a potential area for optimization to improve performance for future work.

In simple terms, for optimal performance, it is desirable for threads within a warp to access contiguous memory locations, ensuring that memory accesses are coalesced. Lastly, in our `gpu-fused`, we eliminated the use of `if-else` as they are the leading cause for warp divergence. We did not observe warp divergence detected in the optimized kernel.

As a result, we fused all the kernels, reduced and shuffled some mathematical FLOPs, and restructured the data layout that is accessed from outside to be coalesced in our optimized version `cuda-fused`. This improves overall `cuda-fused` bandwidth because it has fewer cycles than the other two kernels combined. This is reflected in the total number of cycles per kernel *and execution time* in Figure 2.4, which signifies our improvement in `cuda-fused` by introducing more data reusability in the kernel.

2.3 Optimized Rusanov Solver

As we have shown in our analysis in Figure 2.2, compute kernels suffer from the lowest bandwidth, followed by the updated kernel and then copy kernel `copy-sol`. The compute kernels need to execute FLOPs on the grid data which was copied in the `copy-sol` kernel. Then, the update kernel uses the output of the compute kernels to update the grid with the computed values of the Rusanov evolution in the current timestep. Since those kernels were launched separately, we stored multiple data temporarily, increasing the memory footprint and access to the GM. Thus, if we close the gap between the kernels and *share* the accumulated data between them directly without frequent accesses toGM, we expect to see an improvement in the overall solver's bandwidth.

As a first optimization step, we fused multiple kernels of the copy, compute, and update kernels into a single kernel. This reduced the kernel's launch overhead. Secondly, we eliminated the temporary storage buffers that were used to store intermediate results between different kernels. Since the kernels now can communicate intermediate data using SMEM across different threads within a CUDA block. As a result, we avoid the bandwidth overhead stemming from unnecessary memory transfers from and to GM. We explained earlier in subsection 1.3.2 that frequent accesses to GM impacts significantly the memory throughput of the kernel because it has the lowest bandwidth across other memory types (cf. Table 1.1). Finally, we coalesced the memory access pattern to GM data (cf. Figure 2.7), which increased the spatial and temporal locality of the fetched data from GM to execute the computations depicted in Equation 1.7.

As shown in 3, we utilized CUDA SMEM [9] to store some physical values we retain for later access from each volume. The bandwidth of the shared memory in NVIDIA GPUs is usually much higher than GM access *default location of variables* (cf. [8]). More insights will be provided in the profiling analysis in the upcoming section.

We aligned threads (volumes) within the blocks (patches) to match the input grid Q_{in}. This simplifies mapping the input grid volumes to CUDA threads. A single CUDA

Algorithm 3 Optimized Rusanov kernel algorithm

```
1: procedure EVOLVE(noOfPatches, Q_in, Q_out, newMaxEigenvalues)
2:     $patch \leftarrow blockIndex$    ▷ Threads evolve the same volume index across patches
3:     for $patch < noOfPatches$ do
4:         for all $Dimensions$ do
5:             $s\_maxEigenvalues[currentCellIndex] \leftarrow computeMaxEigenvalue(l\_QOut)$
6:             $s\_flux[currentCellIndex] \leftarrow computeFlux(l\_QOut)$
7:             $sync()$    ▷ Synchronize threads in the block
8:             if current volume is an interior volume then
9:                 for all $unknowns$ do
10:                    $l\_QOut[unknown] \leftarrow$
11:                        $maxEigenvalueInNeighbors(Q\_in, s\_maxEigenvalues)$
12:                    $l\_QOut[unknown] \leftarrow fluxInNeighbors(Q\_in, s\_flux)$
13:                    $QOut[currentCellIndex + unknown] \leftarrow l\_QOut[unknown]$
14:                end for
15:                    ▷ Reduce Maximum Eigenvalue in block
16:                $s\_maxEigenvalues[currentCellIndex] \leftarrow$
17:                    $max(computeEigenvalue(l\_QOut, Dimension),$
18:                    $computeEigenvalue(l\_QOut, Dimension + 1))$
19:                $sync()$    ▷ Synchronize threads in the block
20:                $s \leftarrow$ half the dimensions size
21:                for $s > 0$ do
22:                    if $currentCellIndex < s$ then
23:                        $s\_maxEigenvalues[currentCellIndex] \leftarrow$
24:                            $max(s\_maxEigenvalues[currentCellIndex],$
25:                            $s\_maxEigenvalues[currentCellIndex + s])$
26:                    end if
27:                    $sync()$    ▷ Synchronize threads in the block
28:                    $s \leftarrow s/2$
29:                end for
30:                if $currentCellIndex == 0$ then
31:                    $newMaxEigenvalues[patch] \leftarrow s\_maxEigenvalues[currentCellIndex]$
32:                end if
33:            end if
34:        end for
35:        $patch \leftarrow stride$    ▷ Threads jump a stride of grid x-dimension
36:    end for
37: end procedure
```

thread is a single volume. Where a group of volumes (threads) form a single CUDA block *patch*. This is utilizing the *Halo Threads Concept* that was introduced in [1]. By doing so, we are distributing the workload on the CUDA block equally. This is desirable because different workloads in the warps impact heavily the overall performance. And threads in the same warp that are idle *executed less work* and waiting for other threads *with more work* will result in an underutilized kernel efficiency.

3 Results

In this section, we present our results of running the baseline for this research, which is the base OpenMP implementation[1]. We call it `omp-original` in our results. First, we show the achieved memory bandwidth in our optimized kernel `cuda-fused` versus base cuda kernel `cuda-original` and base OpenMP kernel `omp-original` in Figure 3.1:

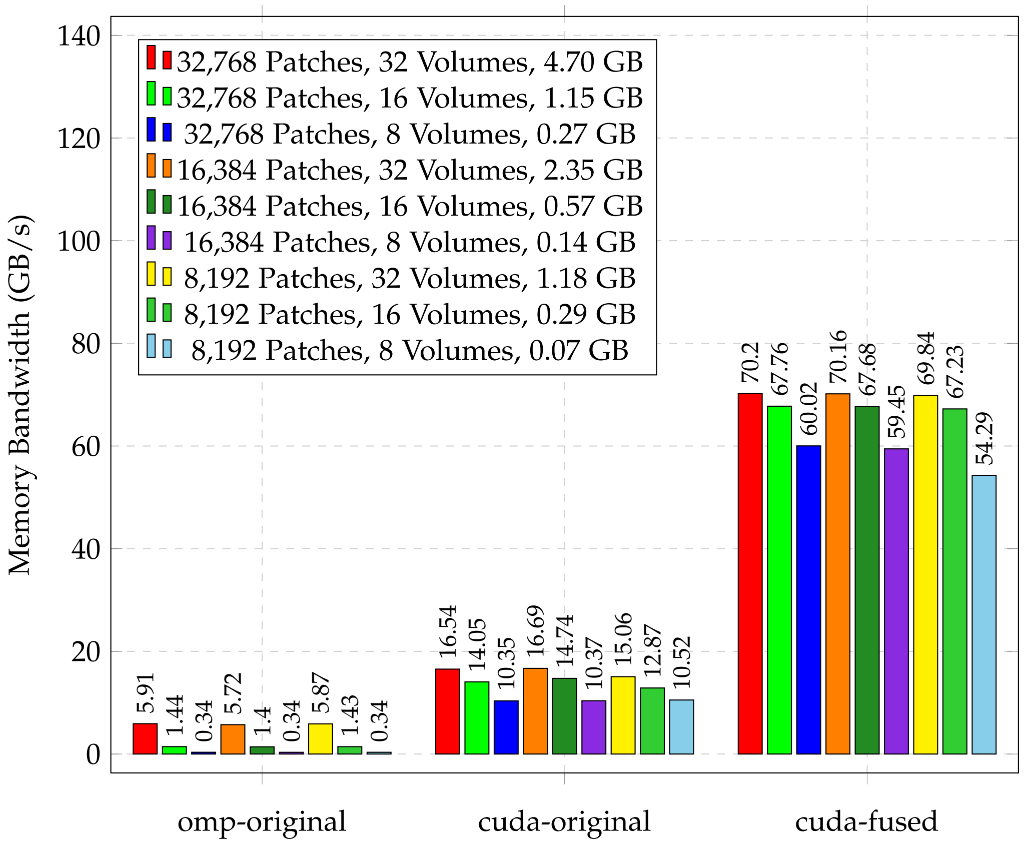


Figure 3.1: Three versions of the Rusanov solver kernels configured with different patch sizes *CUDA blocks* and volumes *CUDA threads*. They employ different computation and memory transfer layouts reflected in the absolute bandwidth.

[1]The base implementation can be found in ExaHyPE2 under tag`openmp-baseline-snapshot`

Our optimized kernel achieves up to $70(GB/s) \times 100/681(GB/s) = 10\%$ efficiency as we explained in Equation 2.8, unlike OpenMP and the original CUDA implementation with separate kernels and intermediate temporary data handling. The complete speedup information is shown in Table 3.1:

Kernel	Peak BW (GB/s)	Speedup (w.r.t. OpenMP)	Efficiency from Peak (%)
omp-original	5.9	1	0.76
cuda-original	16.7	3x	2.3
cuda-fused	70.2	12x	10

Table 3.1: Hardware specifications of our test bench

This improvement is mainly due to the use of shared memory for the intermediate values like the `tmpMaxEigenValue` and `tmpFlux` as we have shown in 3. This, accompanied by coalescing most memory access from GM, helps mitigate a noticeable portion of the memory low throughput we have seen in the OpenMP and CUDA baselines.

We also show the execution time of the three solvers in Figure 3.2, showing that with `cuda-fused`, we have an order of magnitude speedup on the CUDA thread level:

$$\text{Thread Speedup} = \frac{2.71 \times 10^{-8}}{9.75 \times 10^{-10}} = 28x \tag{3.1}$$

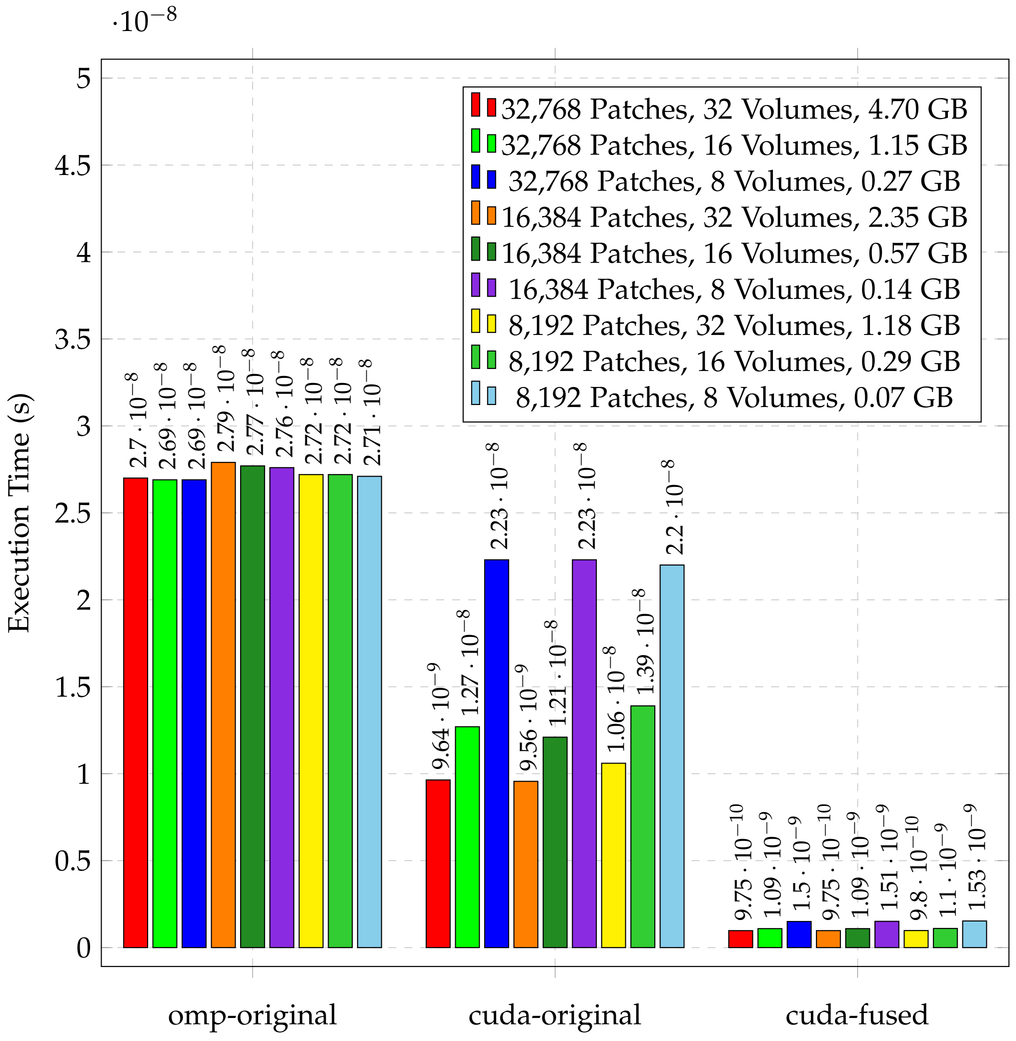


Figure 3.2: Execution time of the threads in the tested kernels. This is the time needed to transfer the grid data to the thread, execute FLOPs on its elements, and copy the solution back to the GM. It contributes to the overall warp efficiency (cf. subsection 1.3.1).

Finally, we plot the efficiency of the solvers' volumes w.r.t. achievable peak bandwidth in Figure 3.3. Although we had a good speedup on the CUDA thread level, we are still far from the achievable bandwidth at 20% as we explained in section 2.2. We are steadily oscillating in the range of 2.4%. This shows that further work could be done to optimize the data access patterns by restructuring the algorithm and reducing memory transfers.

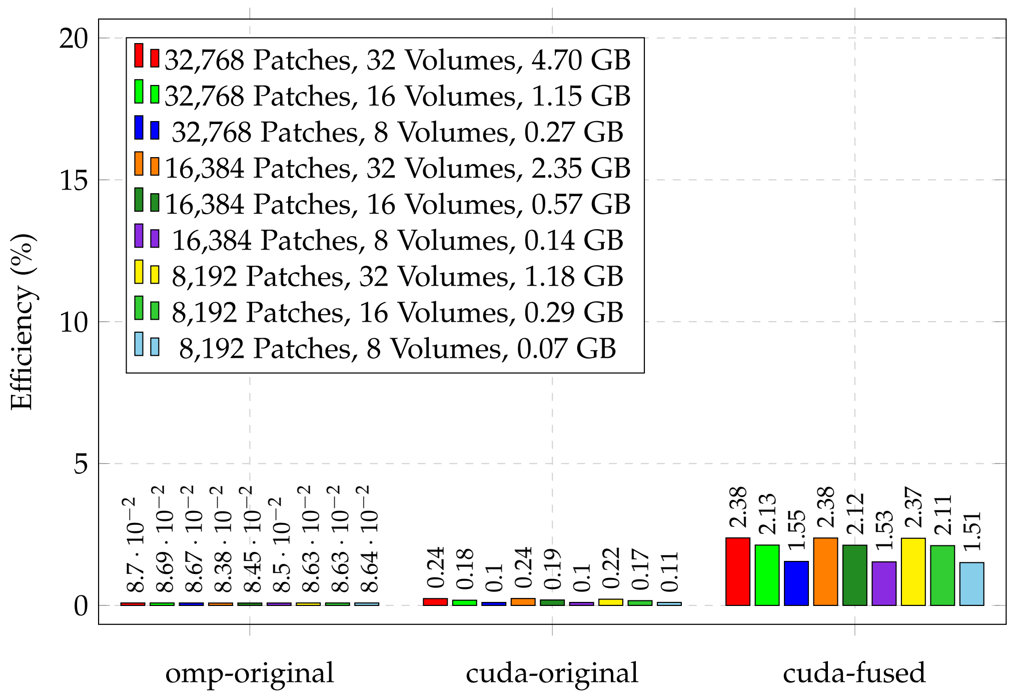


Figure 3.3: Efficiency of the volume bandwidth as a percentage of the reported achievable peak bandwidth valued at 20% of HW measured peak bandwidth of RTX 3080 at 681GB/s that was achieved in reference kernel `copy-kernel` as a baseline (cf. section 2.2).

4 Conclusion and Future Work

We leverage the NVIDIA Nsight platforms for in-depth performance analysis to optimize the implementation of our CUDA kernels used in the Rusanov solver. Post-implementation, an optimization phase was executed, leveraging tools like Nsight Compute for an in-depth performance analysis. This evaluation spotlighted issues, particularly uncoalesced memory accesses and warp stalling caused by memory operations, which we efficiently mitigated by fusing multiple kernels and refining the iteration operations in the time-step solution. These operations, intrinsically dependent on diverse memory portions, were optimized to minimize global memory access. Moreover, we capitalized on shared memory, facilitating the storage of inter-thread data, such as specific physical computations like pressure and maximum eigenvalue. While we achieved a significant reduction in global memory accesses, optimization avenues remain open, particularly in the single-request, single-execution paradigm.

For future endeavors, the spotlight will shift to refining the solution algorithm itself. It is noteworthy that our current progress did not tamper with the fundamental algorithm. The subsequent phase aims to address existing constraints in our `cuda-fused` kernel, such as the restrictions posed by shared memory. This will necessitate reconfiguring the algorithm and reconsidering our approach to propagating spatial and temporal physical values across cells. On another front, taking advantage of the automatic tasking from CUDA is proving to be rewarding. More support for algorithms in the C++ standard is taking pace, allowing more algorithms to be heterogeneously offloaded to the GPUs implicitly without explicit management by the users.

List of Figures

List of Tables

Bibliography

[1] J. X. et al. "Optimizing Finite Volume Method Solvers on Nvidia GPUs." In: *IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 12, pp. 2790-2805* (1 Dec. 2019, doi: 10.1109/TPDS.2019.2926084).

[2] NVIDIA. *CUDA Best practices*. Accessed: 2023-08. 2022. URL: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf.

[3] NVIDIA. *CUDA Graphs*. Accessed: 2023-08. 2019. URL: https://developer.nvidia.com/blog/cuda-graphs/.

[4] NVIDIA. *CUDA Samples Bandwidth Evaluators*. Accessed: 2023-10. 2017. URL: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/1_Utilities/bandwidthTest.

[5] NVIDIA. *CUDA Streams*. Accessed: 2023-08. 2015. URL: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[6] NVIDIA. *CUDA's SIMT Architecture*. Accessed: 2023-08. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture.

[7] NVIDIA. *Nsight Compute Profiling Guide*. Accessed: 2023-09. 2023. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#abstract.

[8] NVIDIA. *Optimization using SM*. Accessed: 2023-09. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory.

[9] NVIDIA. *Shared Memory in NVIDIA GPUs*. Accessed: 2023-09. 2023. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[10] NVIDIA. *What is CUDA?* Accessed: 2023-09. 2023. URL: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2.

[11] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An Engine for Parallel Dynamically Adaptive Simulations of Wave Problems." In: (May 2019).

[12] P. Ullrich, C. Jablonowski, and B. van Leer. "High-order Finite-volume Models for the Shallow-water Equations on the Sphere." In: *Journal of Computational Physics* 229 (Aug. 2010), pp. 6104–6134. DOI: 10.1016/j.jcp.2010.04.044.

[13] M. Wille, T. Weinzierl, G. Brito Gadeschi, and M. Bader. "Efficient GPU Offloading with OpenMP for a Hyperbolic Finite Volume Solver on Dynamically Adaptive Meshes." In: *High Performance Computing*. Ed. by A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse. Cham: Springer Nature Switzerland, 2023, pp. 65–85. ISBN: 978-3-031-32041-5.

Acronyms

CFD Computational Fluid Dynamics. 3

CUDA Compute Unified Device Architecture. 4

FLOPs Floating Point Operations. 13, 14, 16–20, 27, 32

GB Gigabyte. 16, 17, 20, 30, 32, 33

GM Global Memory. 8–11, 17, 19, 21, 23, 25, 27, 31, 32

HPC High-Performance Computing. 1

HW Hardware. 19, 33

ODE Ordinary Differential Equations. 3

PDE Partial Differential Equations. 1–3

SCCS Scientific Computing in Computer Science. 15

SM Streaming Multiprocessor. 5, 7, 9–11

SMEM Shared Memory. 8–11, 15, 16, 19, 21, 23, 27, 36

UMA Unified Memory Architecture. 12