Acceleration of a Heat Diffusion simulation using CUDA

Espen van Beuzekom ist1115340, Einar Bergslid ist1112548, João Miranda ist1112398, Group 10

***Abstract*—In high-performance computing, the goal is to maximize speedup over a serial implementation by exploring parallelism. This paper focuses on accelerating a 2D Heat Diffusion simulation using GPUs, which are well-suited for this task due to the high data-level parallelism as the temperature at each point can be computed independently. The serial computation is parallelized across GPU blocks, significantly reducing execution time. Further optimizations include computing two time steps per kernel launch and distributing the workload across multiple GPUs. The implementation achieved a maximum speedup of 208x in the best case and 25x in the worst case. Results highlight that performance gains highly depend on the simulation parameters, workload characteristics, and the underlying CPU/GPU hardware configuration.**

I. Introduction

CPUS are general-purpose processors designed to handle a wide range of tasks efficiently. This versatility makes them suitable for most applications, though not necessarily optimal for all, particularly data-intensive workloads such as image processing, machine learning, and scientific simulations. These domains often benefit from specialized processors like GPUs, which exploits data-level parallelism — performing the same operations across large input vectors in parallel.

Unlike CPUs, which typically feature a few powerful cores with large caches, GPUs contain hundreds or thousands of simpler cores with smaller caches. CPUs are optimized for instruction-level parallelism using techniques like out-of-order execution, advanced branch prediction, and speculation. In contrast, GPUs prioritize data-level parallelism, leveraging wide in-order pipelines, large register files, and SIMD-like instructions, with simpler control logic.

Additionally, modern GPUs support thread-level parallelism by interleaving many threads to hide latency and data dependencies, whereas CPU cores typically support only one or two concurrent threads via hyper-threading.

II. Methodology

The provided source code was initially a pure CPU implementation, running the heat diffusion calculation in a nested loop structure, iterating through the entire grid. To exploit GPU parallelism, this heat diffusion computation is restructured into CUDA kernels, dividing the work among a hierarchy of blocks and threads.

An important detail about this implementation is that the original source code was modified to initialize both T and T_new matrices. This change was necessary because, when swapping pointers between iterations, the T matrix has wrong values in the first and last rows and columns during odd iterations. This discrepancy leads to different results in the current implementation, which performs two computation steps per kernel launch.

A basic implementation was first developed, where each thread computes the temperature at the next time step for a single point in the grid, with all memory accesses performed in global memory. The blocks are one-dimensional using a row-wise structure. Building on this naive version, a series of optimizations were applied to improve performance. These optimizations are detailed in the following sections.

A. Blocks

The first improvement is partitioning the grid into two-dimensional blocks using a fixed block dimension defined by the macros BLOCK_SIZE_X and BLOCK_SIZE_Y. The reason for the block configuration being two-dimensional is because the data structure also is two-dimensional. Each block contains multiple threads, and each thread calculates the new temperature of a single point in the temperature grid.

This partitioning into blocks reduces the total number of memory access requests by taking advantage of the significant overlap in matrix elements used by threads within the same block. In particular, the inner elements of a block are reused up to five times, while the border elements are accessed twice. Some elements just outside the block boundary are accessed only once. In contrast, in the row-wise structure, elements within the same row are reused three times, whereas elements from the rows above and below are accessed only once. Therefore, the block-based structure offers greater potential for reducing memory access and improving performance.

The block size should be a multiple of 32 in order to maximize performance as this is the number of threads in a warp for NVIDIA GPUs. For most of this project, we decided to use a block configuration of 16x16 as this results in a block size of 256, which is a multiple of 32. Afterwards other block sizes are tested for different grid sizes, see 7.

B. Shared memory

Originally, the implementation repeatedly accessed temperature values from global memory, causing redundant memory access. The heat simulation was optimized by incorporating shared memory to significantly reduce memory accesses, as shared memory is approximately 100x faster than global memory.

In the improved CUDA version, shared memory is leveraged by preloading small blocks of the grid data into shared memory

using the variable **T_shared[]** at the start of each kernel execution. Each thread block first copies its assigned values and the necessary surrounding boundary elements into shared memory. By doing this, each thread within the block can efficiently access neighboring elements directly from shared memory during the computation step. The reduction in global memory accesses significantly improves computational efficiency by decreasing the total memory latency.

The kernel ensures correct execution through synchronization with **__syncthreads()** after the data loading step, guaranteeing all shared memory elements within a block are properly populated before calculations begin. This approach utilizes the spatial locality discussed II-A.

C. Computing 2 steps

To reduce the total number of kernel launches needed to complete the simulation, it is possible to calculate two time-steps of the heat simulation at once. For this to be possible each thread needs to do more calculations per pixel and it needs to access more of the values from the previous step. This is shown in Figure 1.

For the next temperature of one point to be calculated, we originally need the temperature of all neighboring pixels giving a total of five values needed. To do two simulation steps in a single kernel, a total number of 13 values are needed. First we calculate a single timestep for all five values shown in Figure 1(a), then these values are used to calculate the second timestep for the center pixel.

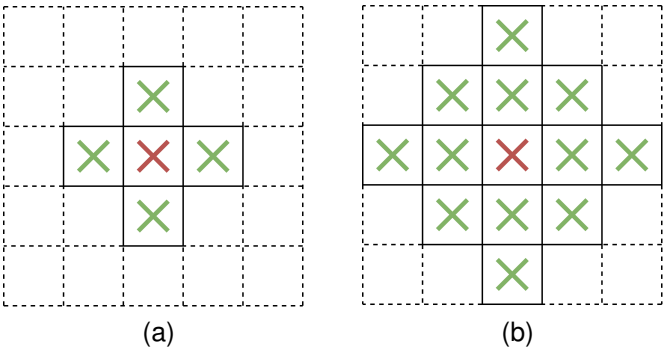

Fig. 1. Values needed from the previous time-step to calculate: (a) 1 time step, (b) 2 time steps per loop-iteration. The center cross indicates the value being calculated.

No thread synchronization is needed after filling shared memory in this implementation. This approach was made because the boundary elements also need to compute the next time-step for their surrounding halo region, which would require adding if statements executed only by these threads. Then, to reduce the number of conditional branches and avoid synchronization, each thread instead computes five values for the next time-step.

Other main issues with this approach involves the elements in rows and columns `1` and `N - 2` of the original matrix, because one of their neighbors are the matrix borders, which temperature never changes, then introducing more conditionals to handle these cases.

Another complication is loading the corner elements of the surrounding boundary, which are also needed to compute updates for some of the halo points. This introduces even more if statements into the code.

Furthermore, if the number of iterations is odd, an additional kernel is launched to perform the final iteration. Since the total number of iterations is typically at least ten thousand, this extra kernel has a small impact on overall execution time. For this reason, it reuses the same kernel described in II-B, without further optimization.

Despite these extra border-case checks and redundant computations, the implementation still achieves significant speedup — primarily because the kernel launch overhead is relatively high, making the performance gains unrolling the time-step loop more significant.

D. Improving Boundary Access

Due to the two-step optimization described in II-C, `8K + 4` elements are loaded into shared memory, where `K` is the size of one block dimension. However, this introduces several conditional statements before the call to **__syncthreads()**, which reduces overall performance.

To improve memory access efficiency, the four conditional statements originally used to load the corner elements of the surrounding boundary were replaced with ternary operators. These determine whether each corner value should be loaded from shared or global memory, depending on the situation. Since these values are used only once, it avoids unnecessary writes to shared memory, optimizing memory usage.

Additionally, instead of relying on each border element of the block to load the required extra elements — resulting in four separate conditional checks (two for the rows and two for the columns, covering both the immediate boundary and the outer halo) — this was replaced by using pairs of rows and columns within the block to cooperatively load those regions. Although some elements are only used once (which might suggest that storing them in shared memory is unnecessary), this strategy enables coalesced memory accesses for columns and eliminates additional conditionals for both row and column computation. As a result, it reduces both the number of if statements and the volume of global memory accesses, improving overall performance.

E. Extra Optimizations

Additionally, some minor optimizations were implemented, resulting in small performance gains. These include computing constants such as $1 - \alpha$ and $\alpha/4.0$ on the CPU and passing the results (two for each constant) as arguments to the GPU. Pinned memory was also used on the CPU together with streams, which is specially important for two GPU implementation. Finally, the matrices were initialized directly on the GPU using a kernel, avoiding the overhead of computing them on the CPU and transferring the data.

F. Split between GPUs

For machines that have two or more GPUs, like Cuda3, it is possible to split the workload among GPUs to reduce

computation time. This is solved by dividing the temperature grid into two large sections, one for the upper half and one for the lower half, and using the CPU to calculate some values in the intersection between the upper and lower half. This is shown in Figure 2. Note that in this case eight rows are calculated by the CPU.

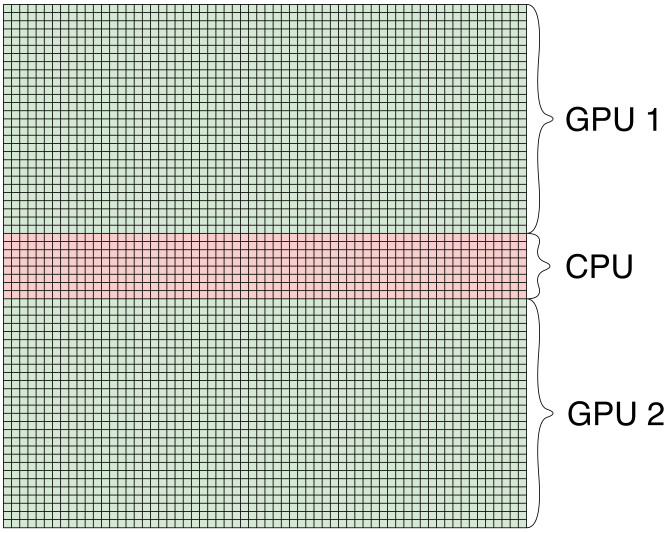

Fig. 2. Temperature grid calculation per device.

In this algorithm, the CPU has `8M` rows, where `M` is a parameter - specifically from `N/2 - 4M` row to `N/2 + 4M` row - while each GPU handles `N/2` rows. However, only the central `4M` rows computed by the CPU (from `N/2 - 2M` to `N/2 + 2M`) are actually used in the final matrix. This means that each GPU computes `2M` redundant rows, and the CPU computes `4M` redundant rows to ensure proper overlap.

Since each kernel performs two time-steps per call, after `M` iterations, the CPU and GPUs exchange `2M` rows to update the overlapping regions. For example, GPU 1 sends rows `N/2 - 4M` to `N/2 - 2M`, while the CPU sends rows `N/2 - 2M` to `N/2`. This approach reduces the communication overhead between devices.

Furthermore, by using four CUDA streams (two per GPU), all four data transfers can be overlapped, allowing them to execute concurrently and minimizing the impact on performance.

Since each GPU processes a different region of the matrix, the border conditions for each one differ. Rather than introducing additional kernel arguments to handle these differences within a single kernel, two separate kernels were developed, one for each GPU with only minor variations. While this approach is more error-prone due to code duplication, it results in a small performance gain by reducing the number of registers used per thread.

III. Results

A. Performed tests

We aim to evaluate the speedup achieved by the application across a range of grid sizes and number of iterations. Additionally, we analyze the impact of different GPU block sizes on performance and compare the execution on one versus two GPUs on the same machine (Cuda3).

For all tests, the default parameters from the Makefile are used, unless specified. Since only multiplications, additions and comparisons are performed on the GPU, the parameters `boundary_row`, α_1, α_2, `T_top` and `T_other` does not affect the execution time, so they are never changed in the tests. We test for five different grid sizes; 512, 1024, 2048, 4096 and 8192. The number of iterations tested were 1K, 10K, 50K 100K and 200K.

We also tested the execution for several block sizes to observe how this could affect performance. The block sizes being used were 8x8, 16x16 and 32x32.

Another simple test conducted was comparing the original (non-GPU) source code compiled with level 2 optimization against the same code compiled with level 3 optimization and Open MP.

An important note is all the GPU speeds ups are relative to the respective CPU of the same machine, and by default the used block size is 16x16. Also, the cuda code was compiled the `-O3` optimization flag.

B. Baseline

Figure 3 and 4 show the CPU execution time for Cuda2 and Cuda3 running the original source code compiled with level 2 optimization. These values were used as a baseline for the calculation of speedup on the respective machines. As expected, Cuda3 CPU has a better performance compared to Cuda2, since it has better hardware specifications.

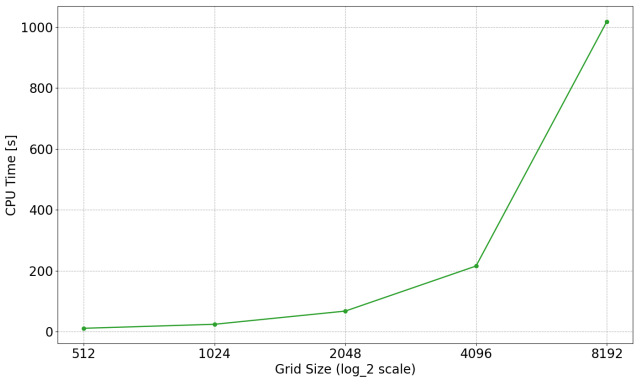

Fig. 3. CPU execution time for Cuda2 using various grid sizes.

For the default parameters we also tested the speedup provided by optimization level 3 and OpenMP. The execution time decreased from 24.0 to 5.1 seconds, resulting in a speedup of 4.7x. Which means that is still possible to have a better usage of the CPU resources

C. Grid Size

Figure 5 shows the speedup when varying the temperature grid size. While both Cuda2 and Cuda3 outperform the CPU baseline, Cuda3 scales significantly better with increasing grid size. For small grids, the speedup difference is smaller, but at larger sizes (e.g., 4096), Cuda3 reaches nearly 185x speedup, whereas Cuda2 drops to around 42x. This can be explained by

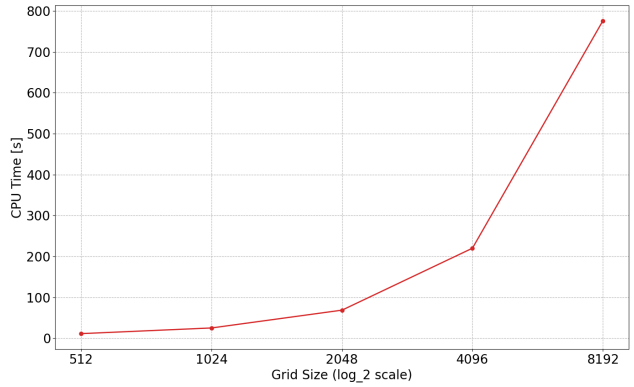

Fig. 4. CPU execution time for Cuda3 using various grid sizes.

the superior hardware on Cuda3, which runs a RTX A4000, while Cuda2 runs a GeForce GTX 1070.

An interesting observation from Figure 5 is that each GPU appears to benefit more from a specific block size, suggesting there is an optimal kernel configuration that maximizes parallelism. This is somewhat counterintuitive, as larger matrices generally offer more data-level parallelism to exploit. One possible explanation for this behavior is that, as matrix rows grow, they may no longer fit in one cache row. Then, accessing multiple consecutive elements could result in multiple cache row writes, increasing memory access latency, ultimately reducing performance.

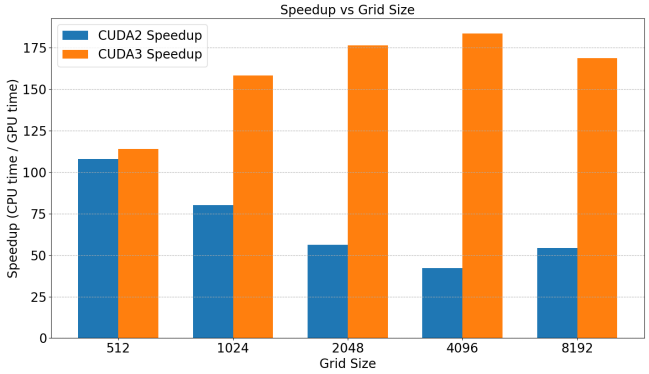

Fig. 5. Speedup for machines Cuda2 and Cuda3 using various grid sizes.

D. Number of Iterations

Figure 6 shows the speedup of Cuda2 and Cuda3 as the number of simulation iterations increases. Both implementations have the same general pattern: speedup improves until 50.000 iterations, and then starts to decline.

The initial increase in speedup (peaking in 50,000 iterations) can be explained by the amortization of overhead. For small iteration counts, fixed overheads such as memory allocation, data transfers and stream initialization (and destruction) dominates the total execution time. As the number of iterations increases, these overheads become negligible relative to the total GPU workload.

However, beyond 50,000 iterations, the speedup begins to reduce for both Cuda2 and Cuda3. This drop is likely due to the increased number of kernel invocations and the changing memory access patterns across iterations. Since the GPU grid scheduler dynamically assigns blocks to cores, it's possible that blocks accessing data already present in the cache — left from recently executed blocks of the same or the previous kernels — will complete faster due to lower memory latency. However, this reuse of the previous kernel data can changes the order of blocks finishing at each run and then switching the scheduling order. In this case, the temporal locality may not be well exploited. As a result, increasing cache misses and, consequently, reducing overall performance after a big amount of iterations.

Once again, Cuda3 outperforms Cuda2, primarily due to hardware differences, as each test is executed on a different machine. The consistently higher speedup observed with Cuda3 indicates more efficient utilization of GPU resources across all iteration counts.

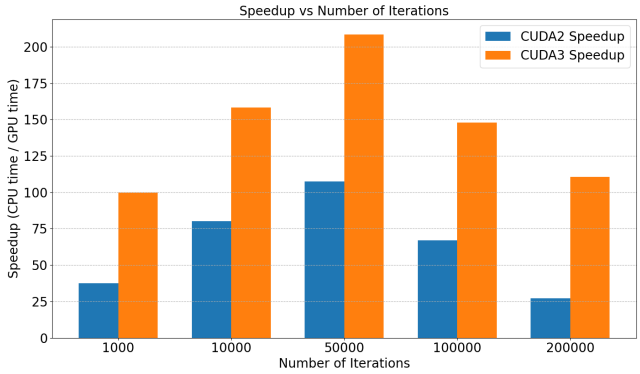

Fig. 6. Speedup for machines Cuda2 and Cuda3 using various number of iterations.

E. Block Size

Figure 7 shows the effect of block size on speedup for different grid sizes for Cuda2. Overall, the performance differences between block sizes are negligible. All three configurations perform almost identically for all grid sizes.

For intermediate grid sizes, especially 1024 and 2048, the 8×8 and 16×16 configurations slightly outperform 32×32. This may be due to smaller shared memory usage (since the padding overhead for load surrounding boundaries elements increases with the block size) or reduced register pressure. Smaller blocks allow for more thread blocks to run concurrently, which can help hide memory latency and improve performance.

However, differences remain small, indicating that the kernel is relatively insensitive to block size within this range. This suggests that other factors — such as memory bandwidth, kernel launch, instruction pattern, or kernel structure — are more dominant in determining performance for this workload.

F. 1 vs 2 GPUs

Figure 8 shows the relative speedup achieved by running the Cuda3 kernel on two GPUs compared to a single GPU. At

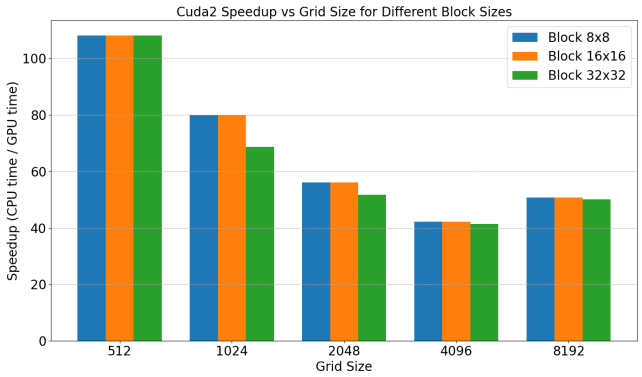

Fig. 7. Speedup for block sizes 8x8, 16x16 and 32x32 using various grid sizes.

small grid sizes (e.g., 512 and 1024), the speedup is close to 1, indicating small benefit from using a second GPU. As the grid size increases, the relative speedup improves, approaching an ideal value of 2 at the largest size tested.

This trend is expected as small workloads do not provide enough parallelism to fully utilize two GPUs, and the overhead of splitting and coordinating work across devices outweighs the benefit. As grid size increases, the amount of parallel work grows, and the GPUs can be kept busier with less coordination overhead per unit of work.

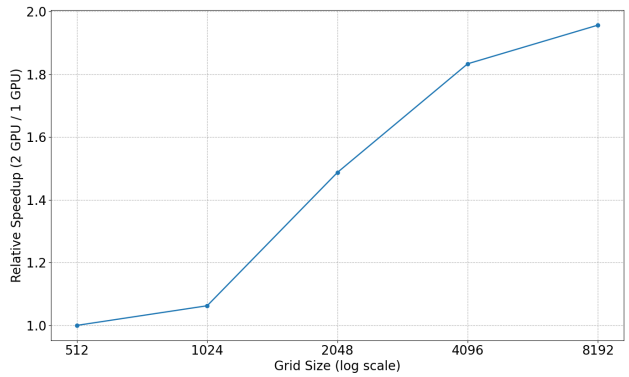

Fig. 8. Relative speedup for 1 vs 2 GPUs using various grid sizes.

IV. Submission Files

All source code is available in a single folder on Cuda3. The path to the directory is `/home/acedes10/heatSim`. As the host and device code is in a single file, the folder contains a one .c file for the original source code provided with the `T_new` initialization and one .cu file for the modified code.

Two new targets were added to the Makefile to make testing easier (`testmultiplegrid` and `testmultipleiterations`). Also, the `debug` and `profile` are added to sanitize memory access and analyze the performance bottlenecks using the NVIDIA tools `compute-sanitizer` and `nsys`, respectivly. Additionaly, the nvcc compiler flag `-Xptxas=-v` was used to verify the number of registers used per kernel.

V. Conclusion

This paper presented a GPU-accelerated implementation of a 2D Heat Diffusion simulation using CUDA, achieving good performance gains over the CPU baseline. By using data-level parallelism and optimizing memory, by using 2D sub-matrix for each block, together with shared memory and two steps calculation per kernel launch, the implementation demonstrated speedups of up to 208× on modern GPU hardware.

Performance evaluation across multiple configurations revealed that grid size and number of iterations significantly influence speedup, while block size slightly influences. Notably, Cuda3 (2 x RTX A4000) consistently outperformed Cuda2 (GTX 1070), due to the better hardware capabilities.

Additionally, we explored the use of multiple GPUs to distribute the workload. While the performance benefit was limited for small grids, ideal parallelism was achieved for larger grids. The experiments also showed that beyond a certain iteration threshold, performance begins to degrade slightly — likely due to cache misses and increased kernel scheduling variability.

Overall, this work demonstrates that careful tuning of kernel structure, memory access patterns, and hardware - aware optimizations can lead to significant acceleration on GPUs.