High Performance Computing for Weather and Climate

Spring Semester 2023

WORK PROJECT 10

Abstracted and low-level GPU programming comparison

Faveo Hörold Thomas Liniger

2023.08.28

Abstract

We implement a number of CUDA kernels to solve a fourth-order diffusion equation. We benchmark each implementation on multiple CUDA devices and relate the performance of each implementation and optimization to the characteristics of the device. We achieve an order-of-magnitude performance improvement over a reference FORTRAN implementation of the code, accelerated with OPENACC. We also investigate the performance of a shared memory implementation and make a fined-grained comparison against our normal implementation using the NVIDIA profiler.

1 INTRODUCTION

The advent of multi-core processors and general-purpose graphics processing units in the last decade has opened the door to more and more available programming paradigms and corresponding speedups in scientific computing.

In this project, we focus on a single numerical micro-task, a specific stencil computation that arises from numerical simulations like computational weather and climate models.

On one hand, exploiting the full capabilities of modern hardware usually involves using programming languages that are closer to the actual hardware implementation, and thus at a lower level of abstraction.

On the other hand, efficient and commercially viable software development requires higher levels of abstraction and therefore higher-level programming languages. Associations of public companies and organizations have tried to close this gap by introducing extensions such as OPENACC to popular programming languages that allow the compiler to exploit some of the parallelization offered by the hardware without putting too much burden on the programmer.

We explore writing a simple discretized stencil operator in CUDA and comparing its performance to a reference FORTRAN implementation accelerated with the OPENACC extension. Starting from the basic CUDA kernel, we implement multiple optimizations and benchmark their performance. We analyze why certain optimizations perform better than others, relating our data to the accelerator architecture. Our implementations, benchmarking setup, and graphing utilities are available online[1].

In Section 2 we describe our optimization strategy. Next, in Section 3, we describe our benchmarking setup. After explaining the numerical micro-task in Section 4, we give an overview of our implementations in Section 5. We graph and analyze our benchmark plots in Section 6. Finally, we study our shared-memory implementation in depth section Section 7 before summarizing our results and giving an outlook in Section 8.

2 OPTIMIZATIONS

Based on the available literature, several optimizations have been proposed to improve the performance of the code. The underlying scheme of many optimizations is to use the memory and cache hierarchy efficiently, either by manually placing data in high-bandwidth memory close to the compute units, or by tiling data in such a way that the hardware's automatic caching strategies are able to deliver data with low latency and high bandwidth. Other optimization schemes target the generated machine code and attempt to reorder or rewrite the source code so that the compute units can run at higher efficiency.

Some of the optimizations require only a rewrite of the code and can then be tested for effectiveness. Other optimizations, such as thread mapping, introduce additional tuning parameters that must be evaluated in a parameter swap, and the optimal value must be carefully chosen. In the worst case, the optimal value is cross-dependent on other optimizations, as in the case of limited resources such as registers or local memory. In this case, the two or more optimizations must be

[1] https://gitlab.ethz.ch/linigert/mpc4wc

balanced against each other to find the best possible combination.

However, the usual optimizations relevant for traditional CPUs may not be effective or relevant in the highly parallel CUDA programming model. We use some of the CUDA device features presented in [SK10] to optimize the performance of our stencil operator. On the one hand, we optimize for CUDA kernel launch latency, but we also remove accesses to memory wherever possible and improve the latency of the remaining memory accesses. In addition to this, we study the effect of varying the number of threads per threadblock, and benchmark a range of working set sizes to characterize the behavior of the CUDA devices.

3 Benchmarking

To benchmark the performance of the CUDA kernel, we use the CUDA measurement facility `cudaEventRecord()` to inject measurements into the CUDA stream just before and after the stencil kernel calls. This effectively measures the time that the CUDA device takes to complete the kernel operations, ignoring any launch and recovery overhead on the host side. We expect that such overhead would amortize for any practical stencil application. Additionally, we use `nvprof` to ensure that the CUDA device is not starved of work at any point in time and to understand how our workload is distributed on the CUDA device.

Figure 1 shows the distribution of runtimes for 100 runs of the optimized kernel. Given that all runs fall within a highly peaked distribution, we only use 3 runs for benchmarking and report the minimum runtime. Overall, the variation between minimum and maximum samples is only about 5%.

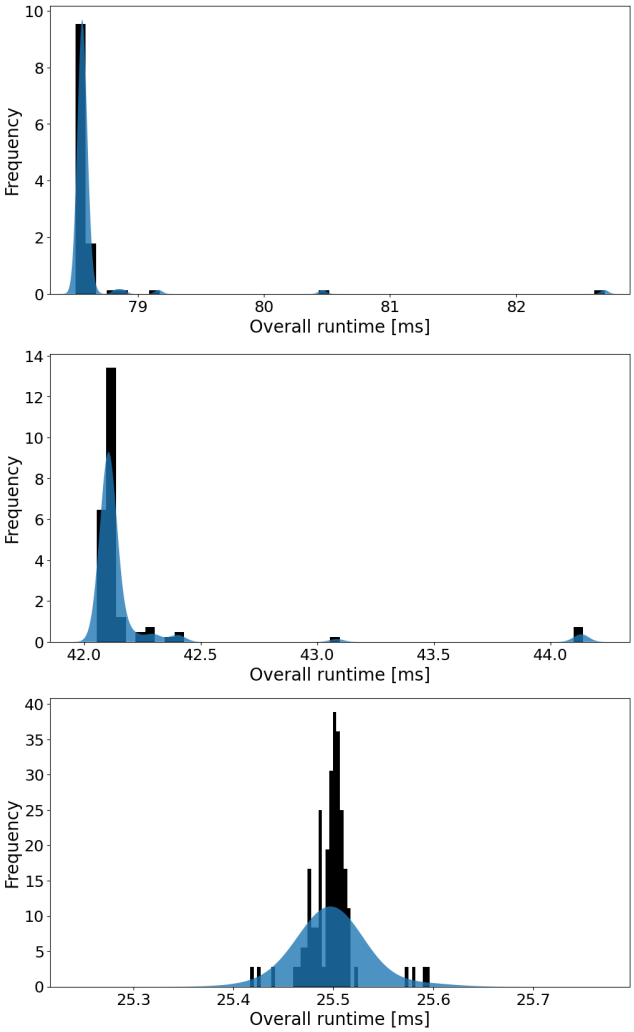

Figure 1: Runtime measurement distribution for 100 measurements on Nvidia P100, V100, and A100 CUDA devices. The distribution of measurements is very peaked, which justifies taking the minimum of a very small number of samples. The overall variation is only about 5%.

4 Stencil computation

We test various CUDA implementations of a solver for the fourth-order monotonic diffusion scheme described in [Xue00]. Such a high-order diffusion scheme may be used in conjunction with flux-limiting in some (atmospheric) codes to dissipate numerical noise. The continuous diffusion equation is written as

$$\frac{\partial \phi(\mathbf{r},\, t)}{\partial t} = S + (-1)^{n/2+1}\, \alpha_n\, \nabla^n\, \phi(\mathbf{r},\, t),$$

where $\phi(\mathbf{r},\, t)$ is the density at location $\mathbf{r}$ and time t, S is a factor for external effects, and α_n is the nth-order diffusion coefficient. We discretize and solve this equation numerically for two-dimensional $\mathbf{r}$. The external factor S is set to zero. Our kernel computes diffusion in multiple layers along the 3rd dimension (z-axis), but there is no coupling between layers. This is similar to a simple atmospheric model, where air tends to stratify into layers, and intra-layer interactions take much more computation time than inter-layer interactions.

To address the fourth-order gradient, we split it into two laplace operators

$$\nabla^2 \left(\nabla^2\right) = \nabla^4,$$

which we discretize using second-order central differencing schemes. In two dimensions, the second-order central difference scheme approximates the Laplace operator as

$$\begin{aligned}\nabla^2 \phi(\mathbf{r}, t) \approx & -4\, \phi\left(\mathbf{r}_\mathbf{L}^{(\mathbf{x},\mathbf{y})}, t\right) \\ & + \phi\left(\mathbf{r}_\mathbf{L}^{(\mathbf{x}-\mathbf{1},\mathbf{y})}, t\right) + \phi\left(\mathbf{r}_\mathbf{L}^{(\mathbf{x}+\mathbf{1},\mathbf{y})}, t\right) \\ & + \phi\left(\mathbf{r}_\mathbf{L}^{(\mathbf{x},\mathbf{y}-\mathbf{1})}, t\right) + \phi\left(\mathbf{r}_\mathbf{L}^{(\mathbf{x},\mathbf{y}+\mathbf{1})}, t\right),\end{aligned}$$

where $r_L^{(x,y)}$ is the discretized position vector at grid point (x, y).

To handle dependence on time, we apply a simple explicit Euler timestepping scheme

$$\phi\left(\mathbf{r}, t_L^{(i+1)}\right) = \phi\left(\mathbf{r}, t_L^i\right) + \phi'\left((\mathbf{r}), t_L^i\right),$$

where t_L^i is the discretized time variable at timestep i. Given that we are not solving a physical problem and are only interested in performance benchmarks, we do not go into detail about any numerical analysis or coefficients. We set $\alpha = 1/32$. We summarize the fourth-order diffusion kernel in Algorithm 1, where the operations are applied to each coordinate x, y, z of the arrays in, tmp, and out.

Algorithm 1 Main diffusion loop

```
i ← iterations
while i ≠ 0 do
    tmp_{x,y,z} ← LAP(in)              ▷ 1st Laplacian
    out_{x,y,z} ← LAP(tmp)             ▷ 2nd Laplacian
    out_{x,y,z} ← in_{x,y,z} − α out_{x,y,z}   ▷ Euler time step
    i ← i − 1
end while
```

The LAP function referenced in Algorithm 1 computes the Laplacian of the input grid, and is given in Algorithm 2.

Algorithm 2 Compute the Laplacian

```
function LAP(in)
    c ← in_{x,y,z}
    l ← in_{x−1,y,z}
    r ← in_{x+1,y,z}
    u ← in_{x,y+1,z}
    d ← in_{x,y−1,z}
    out ← −4 c + l + r + u + d         ▷ Stencil computation
    return out
end function
```

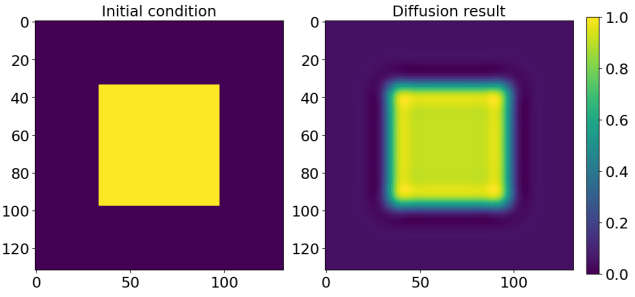

Figure 2: Fourth order diffusion as computed with a square initial condition on a 64×64 grid after 1024 iterations.

We set the domain as a two-dimensional orthogonal grid with periodic boundary conditions. Figure 2 shows the effect of applying the fourth-order discretized diffusion operator on a square initial condition.

5 Implementation

When implementing the stencil computation in a programming language like C,C++, or Fortran, one would typically iterate over the grid points to compute the Laplacians and next timestep sequentially. Such an implementation could then be parallelized by splitting the implementation over multiple cores or processors, for example by splitting independent loop iterations. When implementing the algorithm in CUDA, we are coerced to think about parallelism from the very beginning. Incidentally, Algorithm 1 offers a simple route to parallelized processing. In each step, new grid points are only dependent on grid points computed in the previous step, but not on grid points computed in the current step. This means that grid points are independent of one another during each step and can be computed entirely in parallel. On the flipside, we need to consider synchronization between steps of the algorithm, so that grid data needed to compute other grid points is not overwritten too early.

Periodic boundary conditions. We handle periodic boundary conditions through the use of halo grid points. Each grid has two rows (or columns) of extra grid points that receive the values on the other side of the grid. Thus, stencil operations which mathematically wrap around the domain can be implemented in a regular fashion, behaving no differently at the boundary of the grid than in the center. This means that we must introduce another step into the algorithm, called a halo-update, where halo points are updated with values from the respective opposite side of the domain.

Use of threads. CUDA kernels can be parallelized across blocks of threads, where each thread block contains a fixed number of threads with access to a shared L1 cache and memory section. However, the number of threads per block is limited to 1024, so in most cases implementations need to make use of both levels of parallelization. Given that our stencil application operates on a stack of 2D grids, we partitioned each grid into square blocks with one thread per grid point in each block. As the number of threads per block increases, there should be a tendency to increase performance as well because caches are used more effectively. Additionally, there is pressure to keep assigning registers and allocating memory for new thread blocks. On the other hand, a huge number of threads per block may also overload the shared cache, and some streaming multiprocessors may be starved of work.

Basic implementation. A simple approach to implement the stencil code is to launch a CUDA kernel for each step in Algorithm 1. This is because CUDA kernels launched into the same stream will get executed sequentially by the CUDA device. In addition to the steps shown in Algorithm 1, we also implement the halo exchange with two kernel calls. First, the left and right sides (excluding corners) are exchanged. Then, the top and bottom sides (including corners) are exchanged. By including corner cells only in the second kernel call, we simply exchange rows of cells and avoid implementing any special cases.

Merging CUDA kernels. As it turns out, no synchronization is required between the second call to the Laplacian function and the Euler time step in Algorithm 1. For this reason, we merge the kernels into a single call that computes the second Laplacian and immediately applies the Euler time step. Every kernel call is associated with some overhead latency. One reason for this can be that some execution units

on the CUDA device may be starved of work near the end of a kernel execution, but cannot take on new work because of the implicit synchronization between kernel calls. Thus, merging kernels can reduce the time during which device execution units are idle.

Removing unnecessary array access. Given that the Euler time step only relies on one cell, it is unnecessary to store the result of the second Laplacian in a global array. Instead, it is enough to keep the result in a local variable until it is used by the Euler time step. Merging kernels is also conducive to this optimization. Removing a memory write and read should improve performance, unless this optimization is already done under the hood by the compiler.

Using threads instead of blocks for the halo exchange. Previously, the halo exchange was split over CUDA blocks to avoid the inherent 1024 thread limit. However, splitting the halo exchange over CUDA threads may bring with it a large performance benefit because CUDA threads have access to the same L1 cache. Thus, there are less main memory accesses and the memory bandwidth is used more effectively.

Further optimizations. Finally, we replace the traditional bracketed `<<<>>>` kernel calls with the specialized `cudaLaunchKernel` function. This means that we need to allocate our own CUDA stream and that we need to package function parameters into structures to pass through the function. Doing this should further decrease the latency needed for each kernel call.

Using double-precision floating point numbers. So far, all implementations made use of floating-point numbers. All CUDA devices used for this report have double-precision capabilities, but using double-precision floating-point numbers will likely incur a performance degradation. At the very least, more data needs to be shuffled between the host and device, which will slow down a memory-bound application like this stencil code.

So far, we kept our code simple and monolithic to make comparisons between optimizations possible, but we implement a final version that offers more run-time configuration possibilities. In addition to offering a configurable number of threads, this code also exposes several implementations of the stencil code. One implementation may be run entirely on the host for reference purposes, while another mirrors the final optimized implementation presented above. A third implementation, making use of CUDA shared memory, is presented in the next paragraph. The floating-point precision of all implementations can be changed on the fly.

Making use of shared memory. So far, all implementations made use of global memory on the CUDA devices. However, CUDA offers the possibility of allocating shared memory for each individual thread block, decreasing read- and write latency, and improving throughput. The Laplacian computation depends on neighboring cells, so using shared memory for the stencil application is not trivial. However, our configurable code also exposes an implementation that makes use of shared memory despite this limitation. Similar to halo cells, this implementation implicitly stores adjacent cells in shared memory in order to avoid the need for synchronization between thread blocks. Finally, input and output fields are kept in place via a pointer swap. The use of shared memory promises higher performance due to more efficient memory access.

6 Experimental Results

To understand the performance of our diffusion implementation, we compare it with an optimized implementation written in Fortran and parallelized with OpenACC. Additionally, we verify the correctness of our results using a Jupyter notebook and the NumPy `allclose` function with a relative tolerance of $1e{-}5$ and an absolute tolerance of $1e{-}8$ for single precision floating-point numbers.

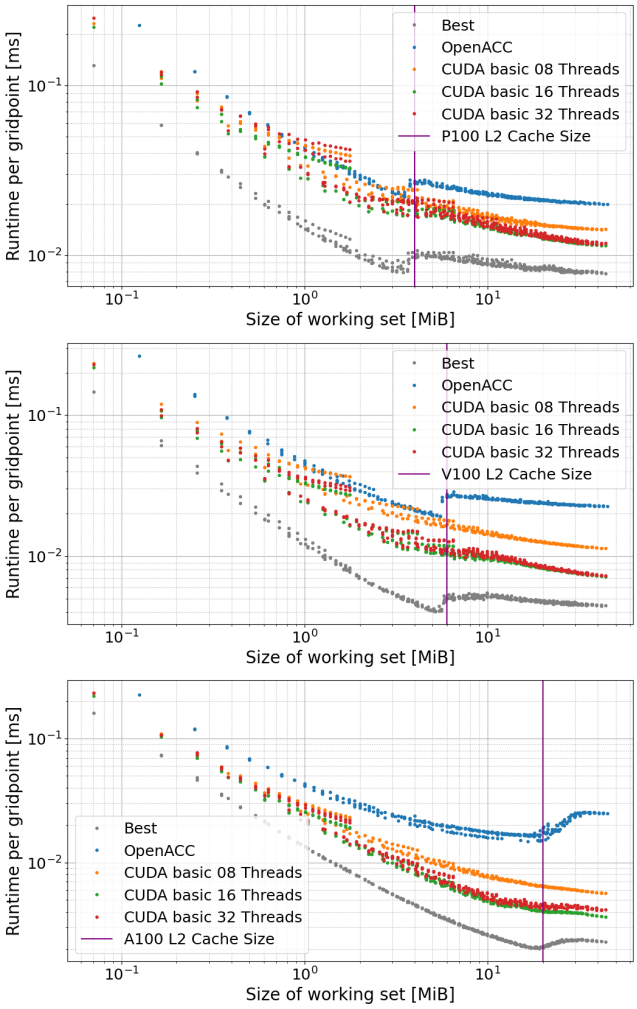

Figure 3: Performance measurements for the basic implementation on P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. Our best result is plotted in gray for comparison. In all three cases, our basic CUDA implementation beats the optimized Fortran implementation. However, only for the A100 does the performance improve over the Fortran implementation for all sizes of working sets.

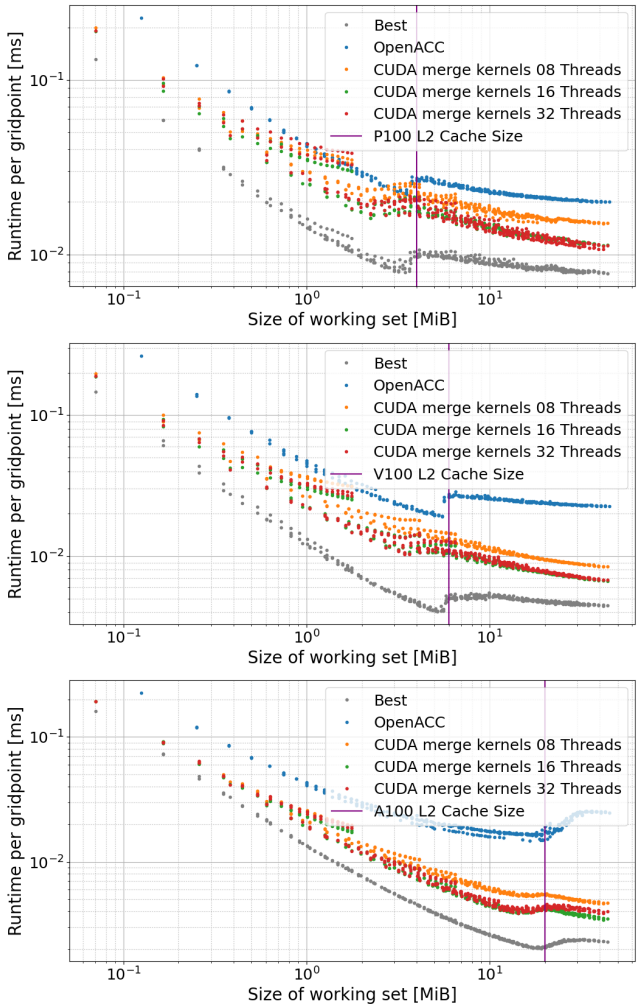

Figure 4: Performance measurements for an implementation with merged kernels on P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. Our best result is plotted in gray for comparison. This implementation shows a small improvement over the previous.

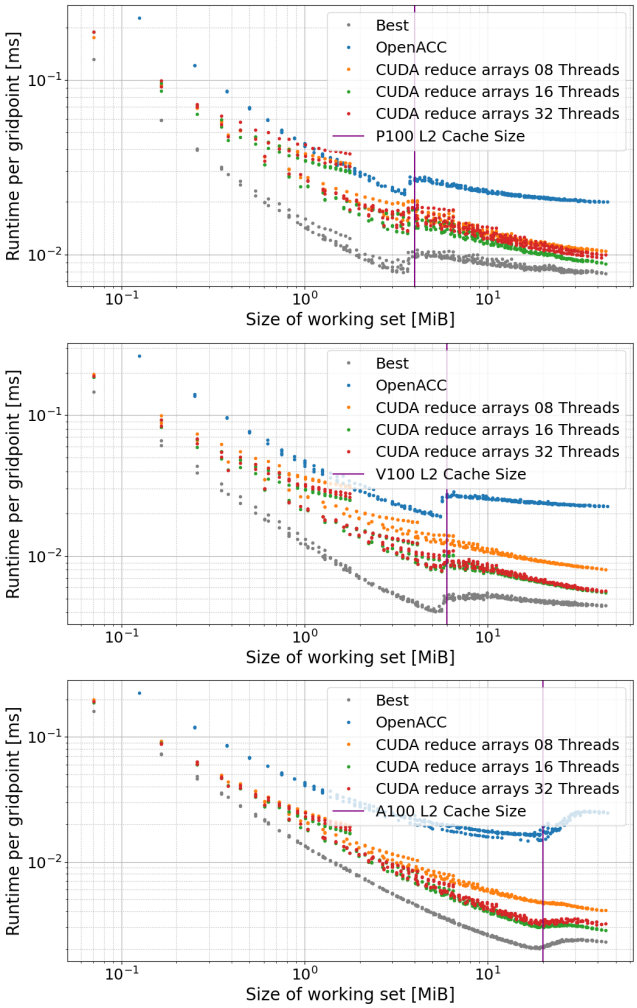

Figure 5: Performance measurements for an implementation where the unnecessary second temporary array is removed, and access to it is replaced with access to a scalar local variable. Plots are for P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. Our best result is plotted in gray for comparison. This implementation further improves on the previous.

We ran our code on a multiple different systems with different Host/Device combinations:

- Nvidia P100 (16GB) hosted by Intel Xeon E5-2690 v3
- Nvidia V100 (32GB) hosted by AMD EPYC 7764
- Nvidia A100 (80GB) hosted by AMD EPYC 7773x

The L2 cache size for each of the CUDA devices is very visible in the performance plots. The P100 has an L2 cache size of 4MB; the V100 has an L2 cache size of 6MB, and the A100 has a huge L2 cache size of 40MB. The L2 cache of the A100 is split into two partitions [Nvi20] of 20MB each. Execution units have slower access to the further L2 cache partition, so 20MB is the more relevant value for this work. As the working set of the stencil grows larger than the size of the L2 cache, there is a characteristic jump in processing time per gridpoint. For ease of interpretation, we mark the size of the L2 cache in each graph. We plot the graph of our best implementation in gray for comparison.

The basic implementation, shown in figure Figure 3, outperforms the FORTRAN reference implementation for most domain sizes. Only for working sets larger than L2 cache do we see a significant improvement over the runtime of the reference implementation. Where the reference implementation working set becomes larger than the L2 cache, we see a dip in performance as more data needs to be loaded directly from main memory. However, our basic implementation does not exhibit such a dip in performance. Rather, our basic implementation exhibits prominent drops in runtime around working set sizes of 1.8MB, 3MB, and 5.5MB. The slower runtimes specifically occur with long and skinny working sets that have a much shorter x-axis than y-axis. The slowdown may be because different memory locations map to the same cache lines, causing frequent cache line evictions and general inefficiency. The benchmarks with 16 and 32 threads show similar performance and are faster than the benchmark with 8 threads. On the A100, the benchmark with 16 threads has an additional edge on the benchmark with 32 threads, especially for large working sets.

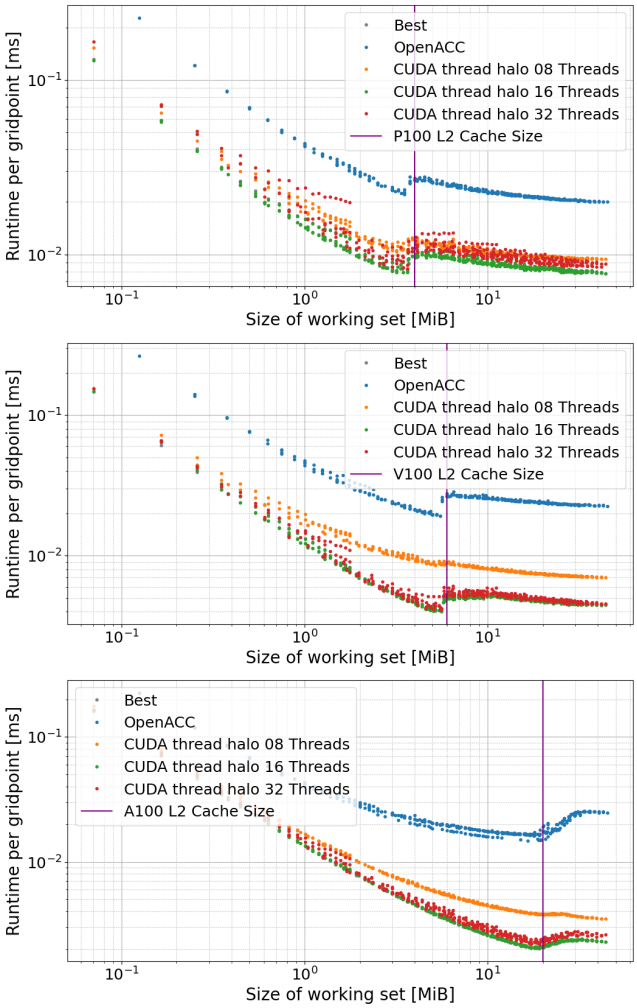

Figure 6: Performance measurements for an implementation where the halo computation is parallelized across CUDA threads instead of thread blocks. Plots are for P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. Our best result is plotted in gray for comparison, but is not visible because it is matched almost perfectly by this implementation.

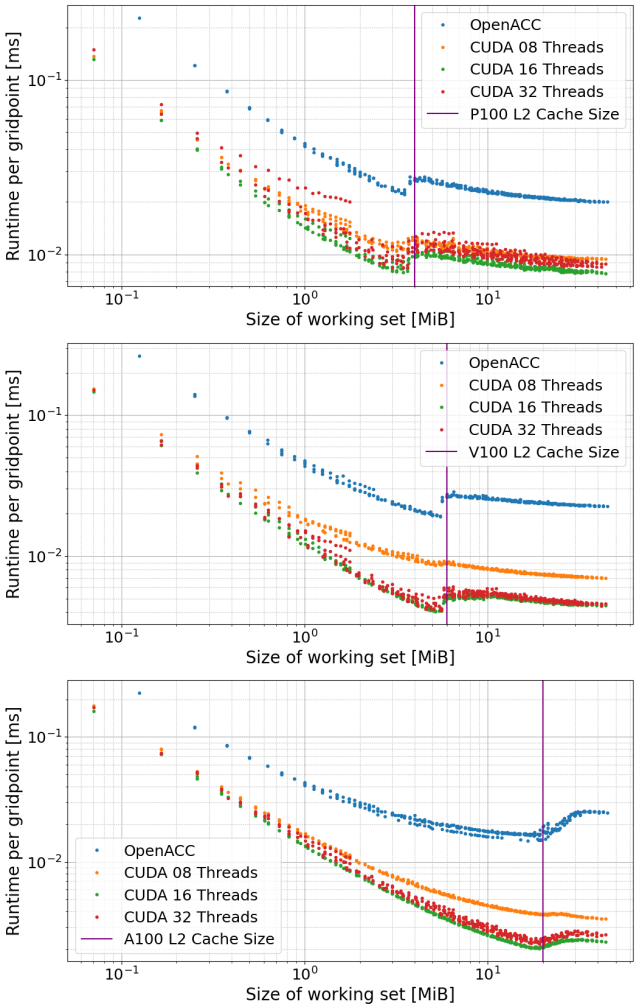

Figure 7: Performance measurements for a fully optimized implementation. We call CUDA device kernels with a specialized function instead of the traditional bracketed <<<>>> call. Plots are for P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. This implementation shows no obvious performance improvement over the last.

Our first improvement to the basic implementation, shown in Figure 4, is to merge the second Laplace kernel with the Euler step. Here, the dip in performance near the L2 cache size becomes more apparent. This means that cache latency becomes more of a bottleneck as the number of kernel calls is decreased. The sudden drops in runtime are also still visible, likely caused by caching effects as described for the basic implementation.

Our next improvement, shown in Figure 5, is to remove the second temporary array. This results in a general improvement in performance. In the previous implementation, the benchmark with 8 threads performed worse than the benchmarks with 16 and 32 threads. However, for the P100, this implementation shows similar performance for all numbers of threads. This may be due to a number of reasons pertaining to the number of active warps, instruction pipeline length, L2 device cache size, memory bus throughput, or memory latency. After this optimization, we see performance approaching our best implementation for larger working sets.

The next step in our line of improvements, show in Figure 6, is to parallelize the halo exchange across CUDA threads instead of thread blocks. With this improvement, our performance has dropped down to the line previously traced out by our best implementation. Especially in the case of the A100, the sudden drops in runtime around 1.8MB, 3MB, and 5MB have become less pronounced, strengthening our case that these artifacts stem from inefficient cache usage caused by strided array access.

We introduce a final set of improvements, shown in Figure 7. The only change made to the implementation was to use the dedicated `cudaLaunchKernel` function instead of the traditional bracketed <<<>>> kernel call. This should reduce the latency of each kernel call, but we see no effect in the resulting graphs. It may be that kernels are already issued quickly enough to the CUDA stream that this latency does not present a bottleneck.

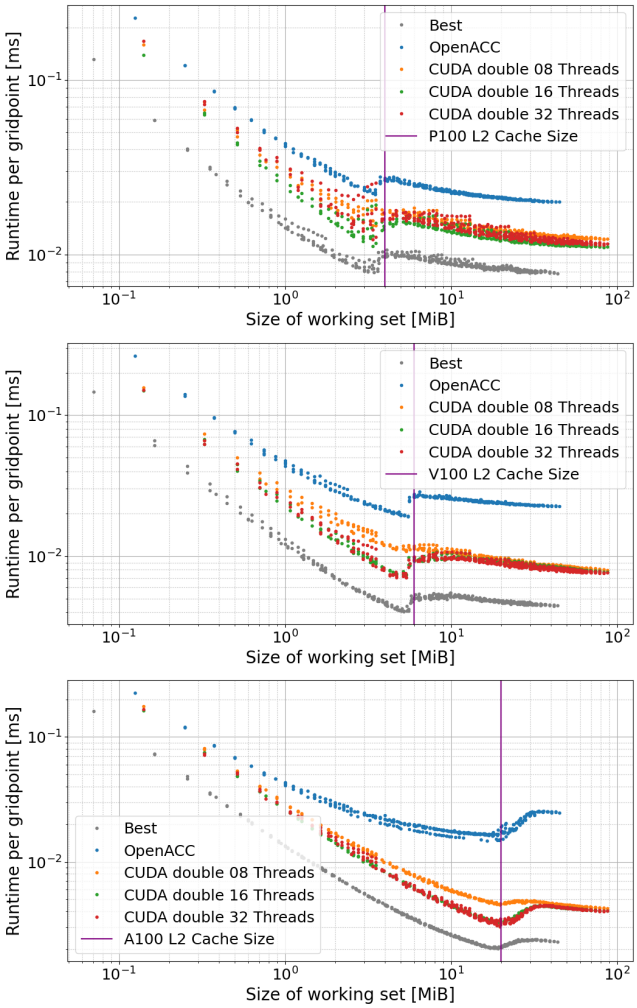

Figure 8: Performance measurements for the fully optimized implementation but with double-precision calculations. Plots are for P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. This implementation shows a serious performance degradation compared to our best implementation.

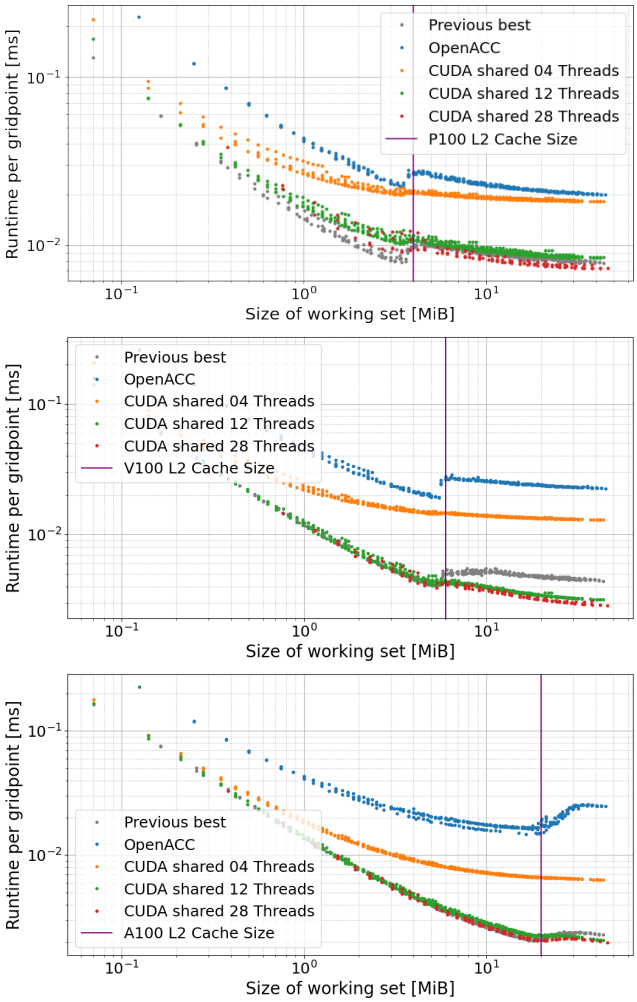

Figure 9: Performance measurements for the shared memory implementation. Plots are for P100, V100, and A100 devices. The L2 cache size is indicated by a purple line. Performance improvements are only observed for working sets larger the L2 cache size.

We also test our optimized implementation with double-precision computations as shown in Figure 8. As expected, there is a large drop in performance, with some instances of the implementation running as slow as the reference implementation on the P100. This makes a strong case for preferring single- or even lower-precision floating-point computations in scientific codes wherever possible.

In a final benchmark, shown in Figure 9, we compare the performance of the shared memory implementation against our previous best implementation. Unfortunately, for working sets fitting in the L2 cache, performance does not improve over our previous best implementation. However, for working set sizes larger than the L2 cache, we observe a slight improvement in runtime. Especially the A100 device seems to amortize main memory latency to the point that performance of the shared memory implementation is essentially identical to that of our previous best implementation. Note that the number of threads are each reduced by 4 compared to previous benchmarks. This to accomodate the 2 extra rows of halo cells around each thread block.

7 Profiling Results

In this section, we analyze the profiling results we obtained using Nvidia's `nvprof` profiler and its `nvvp` analysis tool. The profiler is capable of collecting detailed analysis of a kernel run. To do this, we wrap the kernel call of interest between a pair of api calls `cudaProfilerStart()` and `cudaProfilerStop()`. This is necessary because a full run of hundreds or thousands of kernel calls would overwhelm the GPU profiling bandwidth.

The analysis tool `nvvp` is able to summarize many interesting statistics collected during kernel execution. We will focus on two in particular: the *memory hierarchy access statistic* and the *thread scheduler stall reason statistic*. These statistics nicely show that the bottleneck to main GPU memory is indeed an issue for this specific stencil computation, and that it can be effectively mitigated by using local shared memory on each streaming multiprocessor.

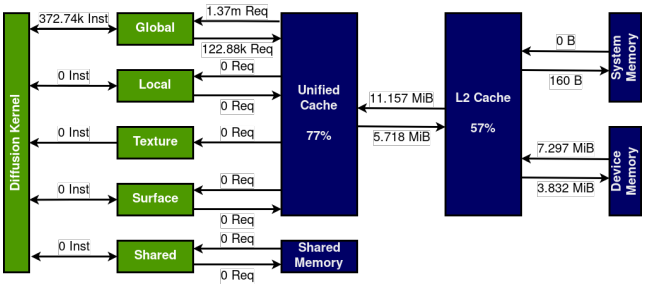

Figure 10: Memory statistics for kernel using only main memory

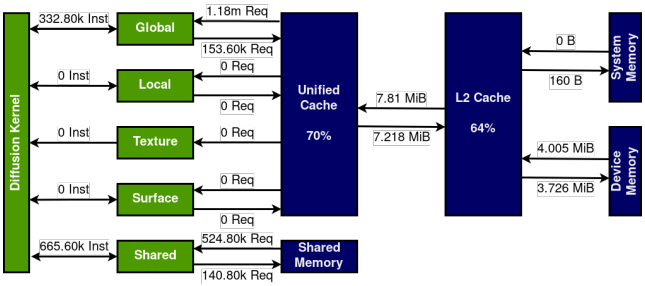

Figure 11: Memory statistics for kernel using shared memory

Memory Throughput and Cache Efficiency

Figure 10 details the load on the various components of the GPU memory system: Green boxes represent logical memory locations, while the blue boxes represent actual hardware units on the GPU. Edges between nodes count the number of transactions, but different measures are used depending on whether the unit is logical or physical. Note that device memory refers to the GPU main memory, as opposed to the host memory connected to the CPU.

From left to right, the numbers on the arrows have the following meanings:

(i) *Coalesced LD/ST instructions.* The first column indicates how many load or store instructions have been issued by the streaming multiprocessors' memory controllers. Each instruction accesses a full 128-byte (512-bit) cache line. Note that the memory controller first collects all memory accesses from the 32 parallel threads in a warp. Then, it coalesces memory accesses to adjacent addresses and finally submits a number of instructions to the L1 cache, given in the graph.

(ii) *Unified cache requests.* The second column shows how many 32-byte requests have been processed by the L1-caches. This is the gross number of memory requests, regardless of whether they could be satisfied by the L1-caches or not. The L1-cache translates the 128-byte instructions into 32-byte requests for the L2-cache. This is why the numbers reported by the L1-cache are counted in 32-byte units. Due to this translation, the sum of read and write requests processed by the L2-cache is exactly 4 times the number of load or store instructions received from the memory controller.

(iii) *Cache hits.* The numbers in the two blue boxes indicate how many memory read requests could be handled by the L1-caches and the L2-cache, respectively.

(iv) *L2-cache and memory throughput.* The remaining numbers indicate how much data is transferred between the L1-caches, the L2-cache, and device memory. Note that not every write to the L2-cache actually causes a write to device memory. This is because the L2-cache buffers write to device memory. In a situation where some data is written to the L2 cache, read shortly thereafter, and then overwritten with new data, the first write request has become dispensable, which reduces the load on the main memory bus.

Comparison between Direct and Shared Memory Implementations

Figure 10 above shows the memory statistics for an implementation which always reads and writes directly to the main GPU memory. Contrast this with Figure 11 below, which shows the same statistics for a kernel which uses shared memory.

We can see that the kernel is now accessing shared memory in addition to global memory. Note that reads and writes to shared memory are segmented differently than global memory accesses. Recall that the memory controllers operate on 128-byte cache lines if global memory is accessed. However, shared memory is accessed in 32-byte memory segments.

The speedup comes from the fact that shared memory accesses are faster than L1-cache accesses. A rough estimate is that shared memory accesses are an order of magnitude faster than L2-cache accesses, and two orders of magnitude faster than device memory accesses.

This is perhaps surprising because shared memory and L1-cache are physically identical, which brings forth the term *unified cache*. However, the memory controllers of the streaming multiprocessors are directly connected to their respective unified caches. Access through the L1-cache takes a different physical path and has significant overhead due to the caching logic. But direct access via shared memory logic has no overhead and is much faster.

On the other hand, the number of global reads and writes has not decreased significantly. This is surprising because the shared memory implementation reads and writes less data from global memory, at least the net number of bytes transferred is smaller.

We have not been able to explain why the number of transactions does not decrease proportionally. One explanation could be that the shared memory kernel implementation uses memory accesses with worse alignment, which means that the memory accesses are misaligned with the 128-byte cache lines. Unfortunately, the memory statistics of `nvvp` do not give more details about how well the memory controller was able to coalesce memory accesses and how much bandwidth was wasted.

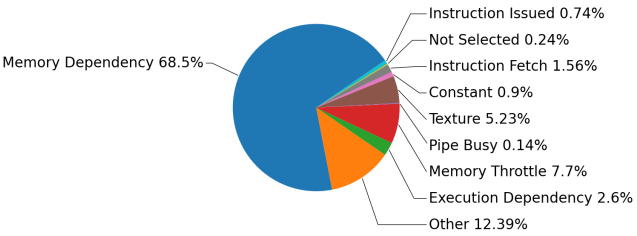

Figure 12: Stall reasons with relative frequencies for kernel using only main memory

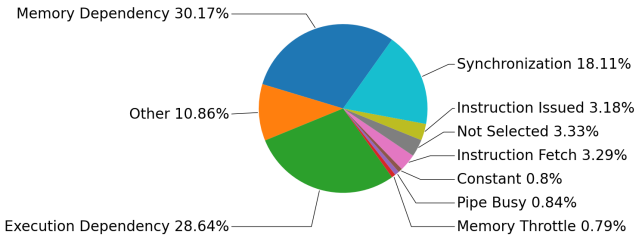

Figure 13: Stall reasons with relative frequencies for kernel using shared memory

Stall Reasons of Warps

To evaluate the effectiveness of the shared memory implementation, we now turn to another interesting statistic collected by the GPU: The profiler periodically checks the state of the active warps assigned to the warp schedulers. Each warp scheduler determines whether a warp is active or stalled, and there are many reasons why a warp might be stalled.

The goal is for a warp to be stalled as infrequently as possible. If a warp does stall, there should be an intrinsic reason for the warp to stall, not an extrinsic delay that could potentially be optimized away.

Consider Figure 12. The plot shows the relative proportion of samples collected during the kernel run, weighted by the different stall reasons. The labels are self-explanatory for the most part, but we will go into more detail about some of the stall reasons, which are relevant in our case:

(i) *Memory dependency.* The warp is stalled because the next instruction is waiting for a previous memory access to complete. On the one hand this could mean that memory far away from the streaming multiprocessor, such as device memory, was accessed. On the other hand, it could mean that the memory coalescing efficiency is low and the memory accesses had to be split into too many requests.

(ii) *Execution dependency.* The warp is stalled because the next instruction is waiting for one or more of its inputs to be computed by earlier instructions. For optimally parallelized code, this type of stall is inherent to the computation and cannot be mitigated. In some cases, loop unrolling or processing multiple elements per thread can help improve execution dependency.

(iii) *Synchronization.* The warp is stalled because it is waiting for all threads to synchronize after a barrier instruction. In our case, this type of stall only occurs for the shared memory implementation. The only way to reduce synchronization stalls is to do as much work between synchronization points as possible, and to keep the workload as balanced as possible across threads.

Comparison between Direct and Shared Memory Implementations

We now turn to the distribution of stalls for the shared memory kernel implementation. We can see in Figure 13 below that the distribution has changed significantly from the previous plot in Figure 12.

First, note that the graph only shows the relative distribution of stalls. It is not an indication of how efficient the kernel is as a whole. The overall efficiency depends on how many instructions are processed and how often stalling occurs to begin with.

The main difference is that stalls due to memory accesses have been greatly reduced and execution dependency has increased. This means that the computing units are now doing more work and waiting for previous instructions to finish, rather than just waiting for data to arrive from memory.

On the other hand, we have to pay a price in terms of additional synchronization stalls. The shared memory implementation reuses shared memory heavily and careful synchronization points are necessary to avoid race conditions.

8 Conclusion

In this report, we introduced a stencil algorithm used for solving a fourth-order diffusion equation. We implemented this algorithm in CUDA and benchmarked several optimizations, comparing the performance of our code to a reference implementation written in Fortran and accelerated with OpenACC. We attempted to relate the behavior of our code with the characteristics of the CUDA devices. By merging kernels, removing unnecessary arrays, and adapting our parallelization strategy, we were able to improve over the performance of the reference code by an order of magnitude. We implemented an additional code that made use of CUDA shared memory to speed up memory access even more. At it's best, this version of our code ran about 1.5x faster than our previous best version. We also used the Nvidia profiler to gain a more fine-grained understanding of how the CUDA device executes our code. Using this data, we were able to explain the benefits and drawbacks of the shared memory implementation.

Future work could continue to investigate the benefits of using shared memory, especially for an algorithm that is more amenable to acceleration through shared memory access.

References

[Hol12] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, 2012, pp. 311–320. ISBN: 9781450313162. DOI: 10.1145/2304576.2304619. URL: https://doi.org/10.1145/2304576.2304619.

[Hos13] Tetsuya Hoshino et al. "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 2013, pp. 136–143. DOI: 10.1109/CCGrid.2013.12.

[Nvi20] *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. Nvidia, 2020.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685.

[Xue00] Ming Xue. "High-Order Monotonic Numerical Diffusion and Smoothing". In: *Monthly Weather Review* 128.8 (2000), pp. 2853–2864. DOI: https://doi.org/10.1175/1520-0493(2000)128<2853:HOMNDA>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/mwre/128/8/1520-0493_2000_128_2853_homnda_2.0.co_2.xml.