APMA2822B Final Project - Persistent Kernels

Patrick Peng

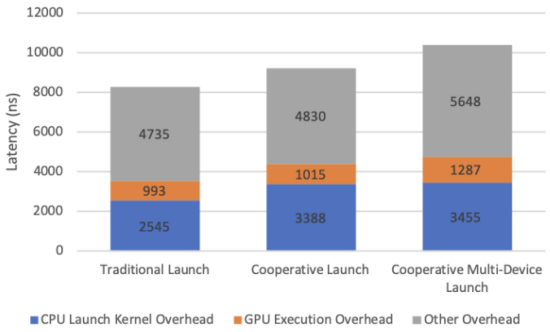

Figure 1: Examples of overheads for launching various types of kernels [1]

1 Introduction

The strengths of GPUs lie in the high degree of parallelism they offer, meaning they can process a large number of instructions and items simultaneously. This makes them very useful for large matrix operations, which arise in fields such as rendering, where the system can pass a large chunk of data to the device and receive processed data later.

However, this efficiency comes at the cost of some latency 1. While this might not be an issue when the problem size is large enough, leading to more time being spent in computation relative to overhead costs, this cost becomes more apparent in smaller problems, and is especially relevant in highly latency-sensitive real-time applications. In this project, I explore this latency cost through the implementation, optimization, and comparison of a persistent kernel with a regular kernel for solving problems of various sizes.

1.1 Persistent Kernels

Instead of launching a kernel every time a task needs to be run, persistent kernels run continuously, communicating with the host to determine what tasks to run and when. This communication system involves both the host and device spinning on shared signal variables while waiting for a signal of completion of a task or a signal to begin a specific task, respectively. The persistent kernel finishes execution upon receipt of a specific signal, instead of completion of a task.

This approach directly avoids many of the overheads associated with launching a kernel for each task, including the launch and execution overheads. It also permits more granular GPU management, allowing you to determine explicitly how blocks or threads are synchronized and what tasks you want to be run on those blocks or threads at what time.

On the other hand, as persistent kernels cannot be modified once launched, the desired tasks must all be implemented together into one kernel. Also, less blocks can be utilized while programming, as the GPU has a limited number of blocks it can support per multiprocessor (and then a limited number of multiprocessors.) Combined with needing to specify synchronization, these factors make the implementation of a persistent kernel significantly more difficult than the standard launch paradigm.

2 Methodology

2.1 Problem Statement

To compare the performance of the two types of kernels, I decided to use the 2D differential equation solver problem from previous exercises. This problem involves iteratively solving for the solution to a given 2D differential equation over a unit square with corners at $(0,0)$ and $(1,1)$. To vary the problem size, I divide each side of the unit square into N points, including $0, 1$. This forms a matrix u of size $N \times N$.

The iterative solution uses the following approximation, where $\Delta = \frac{1}{N-1} = \Delta_x = \Delta_y$:
For a second order differential equation

$$\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} = f(x,y)$$

We estimate $\frac{\partial^2 u}{\partial^2 x}, \frac{\partial^2 u}{\partial^2 y}$ with the second order difference scheme:

$$\frac{\partial^2 u}{\partial^2 x} = \frac{u[i-1,j] - 2 * u[i,j] + u[i+1,j]}{\Delta^2}$$
$$\frac{\partial^2 u}{\partial^2 y} = \frac{u[i,j-1] - 2 * u[i,j] + u[i,j+1]}{\Delta^2}$$

This provides us with an update equation for each matrix element at each timestep:

$$\begin{aligned} f(x,y) &= \frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} \\ &= \frac{u[i+1,j] + u[i-1,j] - 2u[i,j]}{\Delta^2} + \frac{u[i,j+1] + u[i,j-1] - 2u[i,j]}{\Delta^2} \\ &= \frac{u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - 4u[i,j]}{\Delta^2} \\ u[i,j] &= \frac{u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - \Delta^2 f(x,y)}{4} \end{aligned}$$

We also pick an arbitrary solution $\sin(2\pi x) \cdot \cos(2\pi y)$, giving us $f(x,y) = -2 \cdot (2\pi) \cdot (2\pi) \cdot \sin(2\pi x) \cdot \cos(2\pi y)$. After filling in the initial conditions, we can then run the iterative solver.

2.2 Implementation Details

In class, we discussed an optimization that involves first updating $u[i,j]$ where i,j have the same sign, then updating the rest of the elements. Along with being faster to converge, this implementation required less space to store a result matrix (no contiguous elements are ever processed in one step, meaning operations can be done in place) and therefore less overhead in terms of managing and possibly copying data, so the latency effects I wanted to examine would be better isolated from confounding factors.

I also did not perform the difference calculations at each time step, as these required extra synchronization between threads and would prevent me from being to compare the two kernels properly, due to differences in block count and other factors.

Finally, I ensured that both implementations (persistent and base) were using the same number of blocks for any problem size so that they could be properly compared.

2.3 Experiment Procedure

The first step in the experiment was to implement the 2D differential equation solver with both paradigms, and reach a high level of optimization in both cases, which could be determined through effective bandwidth and other metrics that were mostly limited by hardware instead of software implementation. After optimizing the implementations, I compared their performance on various problem sizes from $N = 11$ to $N = 10001$. Finally, I performed additional tests with slightly modified kernels to see if I could separate the different possible causes of latency.

3 Persistent Kernel Optimizations

In this section, I will discuss the optimizations I implemented for the persistent kernel. While optimizing, I used $N = 1001$ as my problem size, as that was close to the number of concurrent blocks I could run on the device, and also provided larger runtimes, reducing the noise from random variance. I will skip the optimization of the base kernel, as we discussed that thoroughly in previous exercises.

3.1 Base Implementation

In my first implementation of the persistent kernel, I implemented the CPU and GPU loops as in this algorithm specification. 1 This implementation took about 2049.3us on average for one iteration.

3.2 Thread Synchronization

In my first implementation, I had two calls to `__syncthreads()`, using it as a barrier while threads were waiting for the task signal from the CPU and while threads were waiting to mark a task as finished. I later realized that this level of synchronization was not needed. Instead of having every thread on the device poll the task signal variable, I could have one thread do this, and then update a shared variable on completion. Then, I would also only need a `threadfence_block()` call to ensure the shared variable propagated to other threads properly before any possible updates. This would also reduce the number of atomic calls, which was a big bonus.

This resulted in a 1.86 times speedup, to an average of 1099.2us per iteration.

3.3 Atomic Memory Order

In the sample code, the atomic loads and stores on the device were under the `SEQ_CST` memory order. At first, I didn't know what this meant, but as I was looking into possible optimizations, I realized that these operations were quite costly. In my implementation, as only one thread was

Algorithm 1 Base Implementation

```
1: procedure CPU TASK COMMUNICATION(task)
2:     for block ∈ cpu_to_gpu do block ←task
3:     pending ← nblocks
4:     while pending > 0 do
5:         for block ∈ gpu_to_cpu do
6:             if block == GPU_DONE then
7:                 pending ← pending - 1
8:                 block ← GPU_WAIT
9:         Sleep(25us)
10: procedure CPU LOOP
11:     cpu_to_gpu ← int[nblocks]
12:     gpu_to_cpu ← int[nblocks]
13:     i ← 0
14:     while i <TARGET_ITERS do
15:         CPU Task Communication (RED_TASK_CODE)
16:         CPU Task Communication (BLACK_TASK_CODE)
17:         i ← i +1
18:     CPU Task Communication (STOP_CODE)
19: procedure GPU LOOP
20:     signal ← cpu_to_gpu[blockIdx.x]
21:     while signal ! = STOP_CODE do
22:         while signal == IDLE_TASK_CODE do
23:             Idle Calculations
24:             signal ← cpu_to_gpu[blockIdx.x]
25:         __syncthreads()
26:         if ThreadIdx.x == 0 then
27:             cpu_to_gpu[blockIdx.x] ← IDLE_TASK_CODE
28:         Switch statement on signal
29:         __syncthreads()
30:         if ThreadIdx.x == 0 then
31:             gpu_to_cpu[blockIdx.x] ← GPU_DONE
```

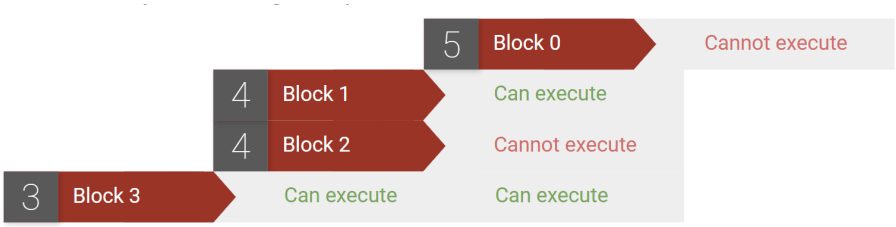

Figure 2: Block 0 is at iteration 5, and must wait for Block 1; Block 1 does not have the data Block 0 needs for the next iteration. However, Block 3 can execute two iterations in a row.

processing the variable for each block, and ordering was not important beyond the basic guarantees of an atomic variable, I realized that I could instead use the `RELAXED` memory order throughout.

This resulted in a 3.57 times speedup, to an average of 307.712us per iteration.

3.4 Block Scheduling

While I was programming the system to communicate with each block, I ran some tests and noted that some blocks were sitting idle while waiting for others to finish the current task. This led me to think of a more granular scheduling algorithm, which would signal blocks to execute as soon as their neighbors were at the same or further steps and the data was ready for the next iteration, with the goal of keeping as many tasks in flight as possible. An illustration of this is in 2.

Although this could result in some blocks being farther ahead than others, and a little bit of numerical instability, I found that this still converged to very similar results to the other implementations. This resulted in a 1.23 times speedup, to an average of 249.864us per iteration.

3.5 Tuning Sleep Duration

This was a simple one–I now noted that my sleep durations were too long relative to the speeds I was now achieving, so I tested a few values and eventually settled on a single `usleep(1)` call in my CPU loop.

This resulted in a 3.61 times speedup, to an average of 69.201us per iteration.

3.6 Iteration Order and Cache Line Accesses

Finally, after the lecture in class on seismic imaging, which emphasized reducing the number of cache lines accessed, I thought I would try to implement this optimization in my code as well. In the $N = 1001$ case, I rewrote my implementation so that each block would process a 27×37 region instead of a 1×999 row. Assuming 128-byte cache lines, this would result in 58 cache lines used per block per iteration instead of 189.

This resulted in a 1.13 times speedup, to an average of 61.22us per iteration.

I thought that this effect might be more pronounced on a larger problem, so I tried to use this optimization on a problem of size $N = 10001$ as well. Instead of each block handling a row and then jumping `gridDim.x` to the next row, I had each block handle contiguous rows. Additionally,

instead of updating row by row, I updated column by column. In this way, once a column was processed the memory from the previous column would not be used again and could be completely dropped. If I were using rows, this data would be needed again on the next row. Similar to the sliding window that was discussed in the seismic imaging lecture, this should result in less cache line accesses.

On the $N = 10001$ problem size, the implementation without this optimization took 3786.46us per iteration; with the optimization it only took 3034.42us per iteration, a 1.25x speedup.

4 Ensuring Correctness

Ensuring the results and algorithms were working as expected was a key part of this project. After each implementation and subsequent optimization, I checked that the truth values at various iterations matched up with other implementations from previous exercises. I also checked all the various iteration loops with a test script to ensure they covered the whole computational domain once per iteration. I also ran multiple tests of each implementation and optimization to ensure stability and consistency of results.

5 Results

5.1 Small Problem Sizes

As the desired latency effect is demonstrated most significantly at small problem sizes, I tested a range of problem sizes from $N = 11$ to $N = 101$. For each of the problem sizes, I ran 1000 iterations to reduce variance due to noise. These results are shown in 3.

At problem size $N = 11$, when there's essentially no computation happening, and very little persistent kernel overhead, the raw cost of executing a kernel (execution overhead) is shown. Note that the kernel launch overhead is mitigated by keeping tasks in flight–the red-black implementation of the task means that all kernels can be scheduled immediately without any memory management in between, so the base implementation (blue in the graph) does not synchronize at each iteration. Then, the difference in average iteration time at $N = 11$ should mainly be due to the cost of moving a block from the queue into execution and then shutting it down, which seems to be on the order of $3 - 4$us.

At slightly larger problem sizes, I assume the overhead from managing blocks and performing a lot of atomic operations begins to make the persistent kernel slower, as the base implementation does not incur much more cost in this regard.

I also wanted to explore the kernel launch overhead, so I modified my base implementation to add a synchronization in each iteration. Then, each iteration would have to start with a launch, and that launch time would not be covered by the computation time in the previous iteration. This implementation's performance is shown in orange in the graph, and from it we can see that the kernel launch overhead seems to be around 10us, possibly a bit less due to the cost of the synchronize call. This would be closer to the performance if we attempted a difference calculation at each step as well, which requires synchronization.

In conclusion, we see the expected results here–the persistent kernel outperforms at small problem sizes due to elimination of the execution and launch overheads.

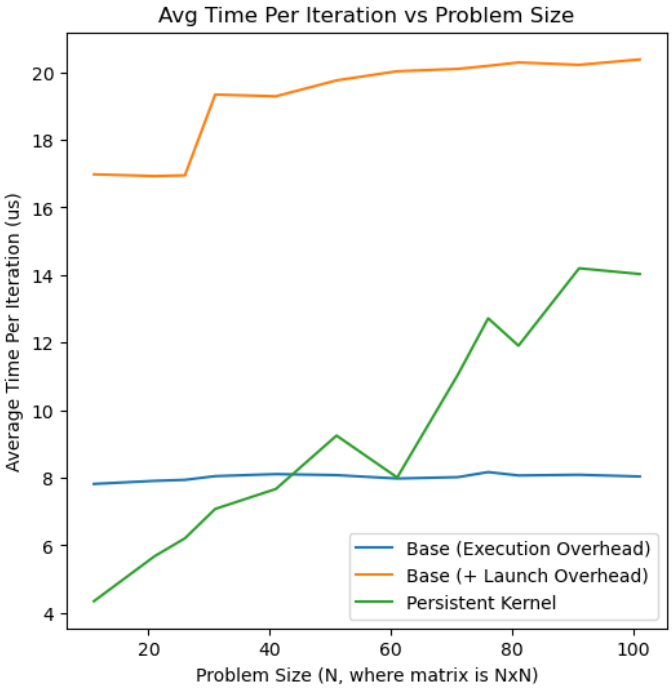

Figure 3: Average iteration time as a function of problem size. Note that the base implementation's average iteration times are mostly constant at these problem sizes, while the overhead for the persistent kernel grows quickly.

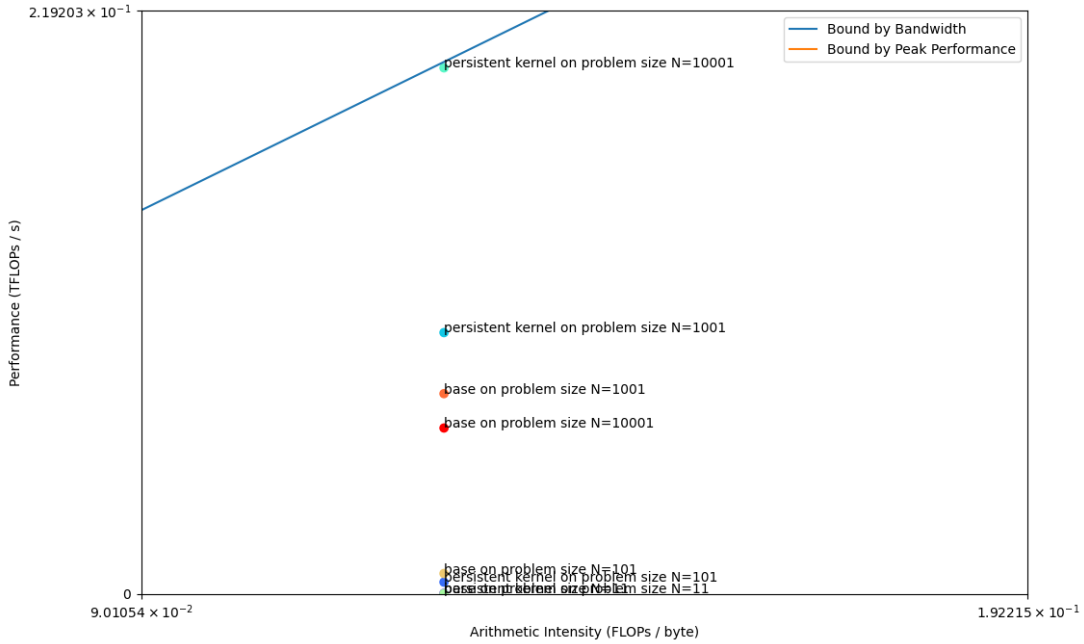

Figure 4: Roofline model, using peak bandwidth of 1.6TB/s and peak double precision performance of 22.6TFLOPs/s. [2]

5.2 Roofline Model

See 4. I do not think the roofline model is very informative for this project, as the goal was to analyze the overheads of kernels and not the pure computational efficiency of the implementation. As such, many of the iteration time results are mostly representing various overheads, making their performance very low on the roofline model.

6 Future Work

6.1 Large Problem Sizes

While testing, I thought I might as well test larger problem sizes, and observed very interesting results, shown in 5. Here, we observe that the average iteration time for the base implementation actually increases faster than the persistent kernel's. I'd like to delve into more of why this is the case–I have a suspicion that it may be due in part to cache line accesses and other optimizations that I did not fully make on the base implementation. However, this does not explain why the growth in average iteration time of the persistent kernel is slower than the growth in matrix size:

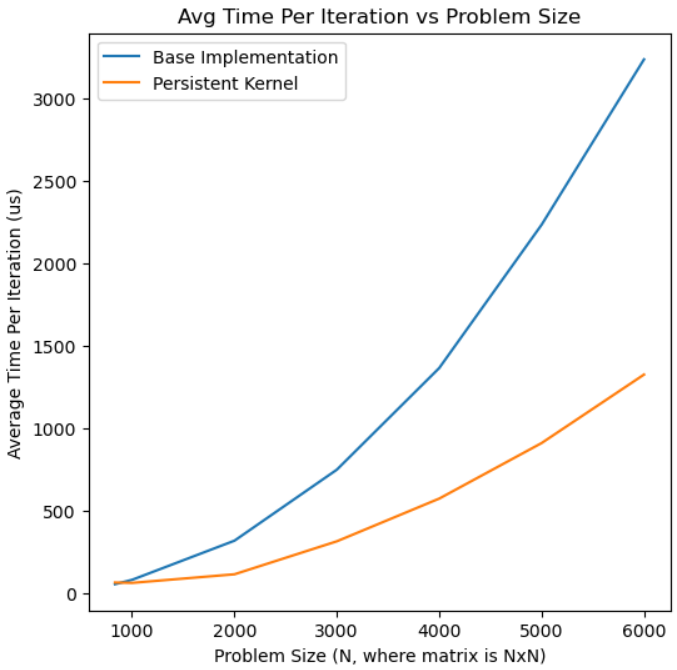

Figure 5: Average iteration time as a function of problem size. Note that the base implementation's average iteration times grow faster than the persistent kernel's with respect to problem size.

for $N = 1001$, my final implementation reached speeds of 61.22us per iteration on average, while for $N = 10001$, the implementation reached speeds of 3034.42us per iteration on average, which is only 49.57 times slower for a problem about 100.2 times larger. The base implementation goes from speeds of 79.9us per iteration on average to speeds of 9628.98us per iteration on those same problem sizes (120.5 times slower), for reference.

6.2 Mitigating Execution Overhead

While researching for this project, I read a post describing how execution overhead could be minimized by streaming multiple smaller kernels to a larger one. This approach seemed like it would be significantly out of scope for this project, but I'm curious if that could bring the performance of a non-persistent implementation closer to a persistent one.

7 Thank You!

A big thank you to Professor Grinberg for taking the time to teach such a unique class, while remaining very practical about it all (I will bring more paper to class in the future.) I also really appreciated his support, especially as I pursued a final project of my own, and his clear passion for the subject, which was definitely infectious. I'd love to do more work in this subject in the future!

References

[1] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. ”Understanding the Overheads of Launching CUDA Kernels” ICPP 2019 (2019).

[2] https://www.amd.com/en/products/accelerators/instinct/mi200/mi210.html