IMPROVING THE PERFORMANCE OF A PARFLOW PROXY APPLICATION THROUGH CUDA PARALLELISM

By

MATTHEW ROBERT ROMERO

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 2 0

Approved by:

Dr. Michelle Strout
Department of Computer Science

Abstract

As hydrological data becomes more in-depth and is measured at higher resolutions, the need for a fast, efficient software framework to perform analysis and simulation grows. HydroFrame is one such framework that allows users to subset inputs and outputs for watersheds in the United States, simulate the movement of water, and analyze the results of their simulations. An important problem to solve within a framework like HydroFrame is producing simulations in a timely manner. Through the effective use of parallel architectures, we can begin to solve this problem. However, with a large codebase, it can be difficult to map out the extensive set of changes necessary. To minimize the scope of the problem and make a bigger difference in a shorter amount of time, the most computationally intensive code is first extracted. After extracting the code, we can begin to test the effects of applying different computing paradigms to it. Multiple variants of the same original code are then compared against each other to determine not only the performance differences, but also the difference in code complexity and practicality. Some variants utilize multiple CPU cores to perform tasks in parallel through OpenMP, while others utilize GPU cores through OpenCL or CUDA. The results show that while parallelism does increase performance on a CPU, it is more effective on a GPU. In terms of the speedup over a baseline serial variant, the OpenMP variant with 8 threads had a 3.57x speedup, the OpenCL variant had a 6.87x speedup, and the fastest CUDA variant had a 11.6x speedup. The metrics gathered show that CUDA offers a far greater increase in performance over OpenCL and should be utilized if a user's machine contains a Nvidia GPU.

Contents

Chapter 1

Introduction

HydroFrame[1] is an important software framework because of the functionality it grants researchers and educators. Through HydroFrame, one can run simulations on watersheds and analyze the results of those simulations. Without the framework, people cannot discover how certain inputs can change watersheds without doing it manually. Doing these experiments manually is costly, time consuming, and adds the risk of permanently altering the existing ecological structure of the watershed. Through simulation, performing the research can be performed cheaply and quickly while still maintaining high precision.

When developing complex software frameworks such as HydroFrame, maintaining high performance across different machines is critical. Those who use HydroFrame expect their simulations to be both quick and accurate. Accomplishing this task often comes after a working version has been developed. During this time, developers will begin to investigate how their code can be refactored and improved upon. In the case of the HydroFrame project, it was found that parallelism could be utilized to increase performance because of the existing structure of the code. However, not every machine can utilize parallelism in the same manner.

Computing resources vary between machines. For example, some machines may have a GPU, which can utilize thousands of compute cores for performant parallel execution. Thus, scientists with access to large supercomputers containing many GPUs would like to utilize the full extent of their compute power by having the simulation code run on GPUs. Those without access to large supercomputers may be limited to CPU execution, where parallelization may still be applied in a different manner. Determining the best parallelization method for a machine's available resources is key in maintaining performance portability.

[1]HydroFrame - https://www.hydroframe.org

Multiple methods of parallelization have been considered for HydroFrame. The framework currently utilizes MPI parallelization (Message Passing Interface), where separate processes send each other messages to coordinate execution simultaneously on a distributed system. For non-distributed systems, utilizing a different parallelization method such as OpenMP can take better advantage of an individual node's resources. OpenMP is a set of compiler directives that allows for threads to operate simultaneously on a shared memory pool. If the user's machine has a dedicated GPU, then it may be better to use a GPU execution library such as OpenCL or CUDA. OpenCL is an open-source library for use on all GPUs, while CUDA is parallel computing architecture used specifically for executing code on Nvidia GPUs. Each new parallelization method is implemented in their own variant of a portion of the existing codebase, where performance metrics are then gathered.

In my approach, I take computationally intensive parts of the existing code base, rewrite the code to utilize CUDA programming models, measure the time it takes for each variant, and compare them with respect to multiple factors. These factors are pure performance improvement over a serial implementation, total lines of source code, and added code complexity. The new CUDA variants I have developed each experiment with memory management and kernel execution in different ways. Base variants that utilized unified memory and manual memory management were initially created. New variants expand upon the base variants by employing new execution techniques, such as wrapping kernels in lambda functions or tiling operations in the z-direction. The following CUDA variants were developed and tested:

1. A base variant that utilizes CUDA's unified memory was developed as a reference point for further developing CUDA variants.

2. A variant that fuses multiple CUDA kernels together into one kernel was developed to measure the effects of minimizing kernel setup and thread synchronization overhead.

3. A variant that wraps kernel calls into C++ lambdas was developed to investigate how kernel setup, execution, and cleanup can be abstracted away through macros that created and executed lambdas on the fly.

4. A base variant that utilizes CUDA's manual memory was developed to measure the performance benefits of using manual memory management over unified memory management.

5. A variant that extracts the members of C structs passed to the kernels into multiple individual parameters was developed to measure the effects of reducing the total number of memory copies between the host and device.

6. A variant that utilizes pinned (page-locked) host memory was developed to measure the performance benefits of faster memory transfers.

7. A variant that tiles domains in the z-direction for asynchronous execution was developed to take advantage of CUDA streams, overlap multiple memory transfers, and minimize memory bus saturation.

Each of the CUDA variants performed better than the OpenMP and OpenCL variants. For example, all CUDA variants had at least a 9x speedup multiplier over the serial implementation, whereas OpenMP had at best a 3.6x speedup multiplier and OpenCL had a 6.9x speedup multiplier. The variants that performed the best were those that utilized pinned host memory (variants 6 and 7). When transferring memory between the host and device, the host memory must first be moved to a page-locked area, then moved over the memory bus to the device. By cutting out the first transfer from unpinned to pinned memory in the host through direct pinned memory allocation, memory transfers were much faster, resulting in faster overall execution. The easiest variants to develop were variants 1-3. Unified memory is very convenient for developers, as memory transfers are abstracted away and performed automatically. Overall, parallelization through CUDA is very performant and should be used over OpenCL if the user's machine contains a Nvidia GPU.

Chapter 2

The LowFlow Proxy Application

LowFlow, developed by Ian Bertolacci, is an application developed to test the effects of each parallelization method considered for HydroFrame on a smaller scale. The application takes a section of computationally intensive C/C++ code from ParFlow (the simulation code in HydroFrame), applies each parallelization method to the code in separate variants, and gathers timing metrics. The section of code contains triple-nested loops that perform certain computations to alter the state of four output arrays. LowFlow makes the development of new variants easy by decomposing the allocation of memory, the data types created, and the execution of code into separate components, thus allowing the developer to target individual concerns when creating or tweaking a variant.

2.1 Serial Variant

The serial variant is a baseline for all other variants as it is implemented to operate sequentially, not in parallel. By not utilizing parallelization methods, the timing metrics for this variant are the slowest. These metrics are used when determining the speedup multiplier for every other variant tested. This variant contains approximately 340 lines of code, which is important in comparing the development complexity between variants.

In this variant, memory allocation is performed through C's `malloc` function. A total of twenty input arrays (which represent data in a three-dimensional grid) are allocated before being passed into the code as separate parameters. The code is then executed in a purely sequential manner. After the code has finished executing, four output arrays contain the results. The output arrays are assumed to contain the correct data for any grid size, and are used to verify the correctness of other variants. Below is an example of the triple-nested loop headers used in computing the output grids.

```
for (int x = 1; x < domain->nx - 1; x++) {
    for (int y = 1; y < domain->ny - 1; y++) {
        for(int z = 1; z < domain->nz - 1; z++) {
            ...
        }
    }
}
```

Figure 1: An example of the loop headers in the serial variant

2.2 OpenMP Variant

OpenMP [1] is a set of compiler directives that allow for parallel execution. Loop parallelization is done by splitting loop iterations across multiple threads that all run concurrently. One can set the number of threads to be spawned through API calls or by setting certain environment variables. In my testing, performance metrics for 1, 2, 4, and 8 threads were gathered. This variant contains approximately 300 lines of code.

The loop headers shown in Figure 1 are able to utilize OpenMP parallelization well, as the associated loop bodies have no race conditions when distributing load across threads. OpenMP provides an easy to use, single line compiler directive that allows the loops to run in parallel. By adding the `private` directive to the pragma statement, each thread spawned will have its own copy of the loop iterators, instead of having to share them. Below is an OpenMP parallelized version of the code in Figure 1.

```c
int x, y, z;

#pragma omp parallel for private(x, y, z)
for (x = 1; x < domain->nx - 1; x++) {
    for (y = 1; y < domain->ny - 1; y++) {
        for(z = 1; z < domain->nz - 1; z++) {
            ...
        }
    }
}
```

Figure 2: Loops parallelized with OpenMP

2.3 OpenCL Variant

OpenCL [3] is a programming model for running applications in a parallel manner. It is designed to be able to run on CPUs, GPUs, and other types of hardware. In my testing, OpenCL is used to run code on a GPU. The benefit of using OpenCL is that it is not specifically tied to one architecture or brand, unlike Nvidia's CUDA APIs. By utilizing a GPU, the performance metrics start to improve over serial implementations as GPUs contain hundreds or thousands of smaller compute cores that allows for parallel execution on a larger scale.

While OpenCL is often faster than serial implementations, it is a complex framework to work with. The source code for this variant required approximately 1,000 lines of code. As a comparison, both the serial variant and the simplest CUDA variant required approximately 340 lines of code. The large number of lines of code is mainly a result of the setup required before execution.

Chapter 3

CUDA Contributed Variants

CUDA [2] is a parallel computing architecture and set of APIs for writing code that will execute specifically on a Nvidia GPU. It provides methods of allocating memory on both the host and device, copying memory between the host and device, and queuing the execution of C/C++ code on a GPU. My contribution to the LowFlow application is the addition of several new CUDA variants. These variants were developed with the intent of comparing the performance metrics between OpenCL and CUDA, as both parallelization methods are GPU oriented.

The new variants can be split into two groups: unified memory and manual memory management. With unified memory, a single API call results in a pointer that can be referenced in both host and device code. Unified memory makes transitioning from host code to device code easy, as minimal changes are necessary to change allocation and data transfers. Manual memory management is more involved than unified memory. Manual memory allocation results in an additional pointer being created as opposed to a single pointer in unified memory. There is typically a host pointer and a device pointer, where data is transferred through CUDA API calls. Though tedious, there are many advantages to using manual memory, such as performance benefits and a higher level of control over execution. The variants that utilize unified memory were implemented first to establish a development pattern for the manual memory variants that were developed next.

3.1 Base Unified Memory Variant

The base unified memory variant was the first CUDA variant developed. It serves as a template and baseline for the other CUDA variants. It was the simplest variant to implement, as all that was necessary was to pull the loops into kernels and change memory allocation to utilize the unified memory system. This variant had approximately 340 lines of code.

In the serial variants, memory allocation is performed by C's `malloc` function. In this CUDA variant, memory allocation is performed through a call to `cudaMallocManaged`. This allocation takes in a pointer to a pointer of the type being allocated, as well as the number of bytes desired. After successful execution, the pointer passed in will contain a pointer that can be referenced on both the host and device. As with the other variants, there are 20 grids passed in, as well as 3 temporary grids created during execution for a specific portion of code, all of which operate within the unified memory system. The similarity between `malloc` and `cudaMallocManaged` allow for easy translation from serial code to CUDA code.

Device code must be pulled into kernels, which are functions used specifically for execution on the device. Execution of kernels is different than normal C functions. When calling a kernel, the developer must specify the CUDA thread grid and block size to be used. A thread block is a three-dimensional collection of threads that may have up to 1,024 threads in total [4]. A grid is a three-dimensional collection of blocks which may have up to 65,535 blocks in any dimension. By specifying the correct amount of threads per block and blocks per grid dimension, loops can be parallelized. Figure 3 shows how the triple-nested loops shown in the serial code are parallelized through CUDA kernels.

```
__global__ void kernel(Grid *grid1, Grid *grid2, ...) {
    //Get current thread indecies, offset by 1
    int x = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int y = blockIdx.y * blockDim.y + threadIdx.y + 1;
    int z = blockIdx.z * blockDim.z + threadIdx.z + 1;

    //Loop body
    ...
}

//Determine thread block and grid sizes
dim3 blockSize = dim3(16, 16, 4);
dim3 gridSize = dim3(domain->nx / 16,
                     domain->ny / 16,
                     domain->nz / 4);

//Call the kernel
kernel<<<gridSize, blockSize>>>(grid1, grid2, ...);
```

Figure 3: Loop parallelization performed through CUDA kernels

3.2 Fused Kernel Variant

The fused kernel variant is a slight modification to the base unified variant. In the code for the base unified variant, there are five total kernels. After investigation, it was discovered that the first four kernels use the same setup, and could be fused into one long kernel. This was as simple as moving the loop body code from each kernel into a singular kernel with an extended list of parameters.

The main purpose of this variant was to test the performance differences between multiple small kernels and one large kernel. The initial hypothesis was that less kernel calls would result in less overall setup, which would increase performance by a moderate amount. However, this was not quite the case, as the performance differences between this variant and the base variant are very small. This variant's main advantage over the base variant is its reduction of the total lines of code, only containing approximately 300 lines.

3.3 C++ Lambda Variant

The C++ lambda variant utilizes a combination of C++ lambda functions, macros, and unified memory to abstract away the setup necessary to run code on the device. The idea was that code complexity could be reduced by removing the need to define kernels manually. In this variant, kernels are created and executed through a macro. This allows the developer to simply copy the loop body into a single macro invocation. The current ParFlow source code utilizes macros to define loops. Thus, the lambda variant could potentially be incorporated back into the source code. Additionally, the developer does not have to worry about a long list of parameters passed to the kernel, as the lambda function can be defined to capture all external variables from the enclosing scope.

The kernel creation is possible through the definition of a kernel "wrapper". The wrapper is itself a normal CUDA kernel that takes in a lambda function. When called, the wrapper will execute the lambda function passed in, which in turn executes the code passed into the macro. Figure 4 below shows the definition of the lambda macro. Figure 5 shows the kernel wrapper and how the macro is used. Through the use of macros in conjunction with lambdas, the code is simplified to approximately 250 lines of code. In addition to containing fewer lines of code, this variant performs approximately 1.5% better than the base variant. This is likely a result of compiler optimizations, such as the possible inlining of some lambda functions.

```
#define Run_Kernel(domain, x, y, z, body) {                    \
    //Determine thread block and grid sizes                    \
    dim3 blockSize = dim3(16, 16, 4);                          \
    dim3 gridSize = dim3(domain->nx / 16,                      \
                         domain->ny / 16,                      \
                         domain->nz / 4);                      \
                                                               \
    //Call kernel wrapper to execute lambda function           \
    wrapper<<<gridSize, blockSize>>>([=] __device__ {          \
        //Get current thread indecies, offset by 1             \
        int x = blockIdx.x * blockDim.x + threadIdx.x + 1;     \
        int y = blockIdx.y * blockDim.y + threadIdx.y + 1;     \
        int z = blockIdx.z * blockDim.z + threadIdx.z + 1;     \
                                                               \
        //Execute code body passed into macro                  \
        body;                                                  \
    });                                                        \
}
```

Figure 4: Definition of the lambda macro used in kernel creation abstraction

```
//Wrapper definition
template <typename Function>
__global__ void wrapper(Function kernel) {
    kernel();
}

//Macro invocation to run code on device
Run_Kernel(domain, x, y, z, {
    //Kernel body here
    ...
})
```

Figure 5: Definition of the kernel wrapper and example macro invocation

3.4 Base Manual Memory Variant

The base manual memory variant was the first CUDA variant to utilize manual memory management and servers as a template and baseline for other variants that also utilize manual memory management. As stated before, manual memory management is more involved than unified memory management. This variant has approximately 460 lines of source code, which is a result of performing memory allocation and transfers manually. The convenience of having a single pointer for both host and device memory is replaced with precise control over how and when memory is allocated or transferred. Additionally, using manual memory opens up exclusive opportunities, such as asynchronous execution.

In this variant, memory allocation is performed through `cudaMalloc`. The function takes in the same parameters as `cudaMallocManaged`, but the resulting pointer only contains device memory addresses. After allocation, there is a host and device pointer, where memory transfers occur through special functions. The function `cudaMemcpy` will transfer memory between the host and the device by taking in a destination pointer, a source pointer, the number of bytes to be copied, and the kind of transfer (an enumerated value specifying the direction of transfer).

When transferring custom structs between the host and device, multiple transfers may be required. In the extracted code, the grids that are passed in contain not only the array associated with the grid, but also information about the domain of the grid. The array is dynamically allocated, so two separate memory allocations and transfers are required: one for the overall struct and one for the array. Figure 6 shows the method used to transfer structs. Using a temporary grid that holds pointers to device memory, then copying those pointers over to the device allows for structs with dynamically allocated members to be transferred to the device.

```
void transferGrid(Grid *host, Grid *device) {
    //Create local struct to hold temp values
    Grid temp;

    //Get memory copy sizes
    Domain *domain = host->domain;
    double *data = host->data;
    size_t domainSize = sizeof(Domain);
    size_t dataSize = domain->nx * domain->ny * domain->nz *
                      sizeof(double);
    size_t gridSize = sizeof(Grid);

    //Allocate memory on device, save pointers into temp
    cudaMalloc(&temp.domain, domainSize);
    cudaMalloc(&temp.data, dataSize);

    //Copy the host data to the device using temp pointers
    cudaMemcpy(temp.domain, domain, domainSize,
               cudaMemcpyHostToDevice);
    cudaMemcpy(temp.data, data, dataSize,
               cudaMemcpyHostToDevice);

    //Allocate device grid struct
    cudaMalloc(&device, gridSize);

    //Copy pointers in temp to device
    cudaMemcpy(device, &temp, gridSize,
               cudaMemcpyHostToDevice);
}
```

Figure 6: Transferring structs between the host and device

3.5 Extracted Struct Members Variant

The extracted struct members variant was developed to test the effects of reducing the number of calls to `cudaMemcpy` by passing individual struct members as separate parameters to kernels instead of complete structs. As shown in Figure 6, three memory transfers are required to transfer a struct with two members. By extracting the members from the struct and passing them to kernels separately, only two memory transfers are required per grid.

The disadvantage to extracting struct members into separate parameters is that there are twice as many grid variables to keep track of. Additionally, there are approximately twice as many parameters to the kernels. This can make debugging difficult if one of the parameters is out of order. If a kernel previously referenced a struct, it will need to be rewritten to instead reference the name of individual members. This variant had approximately 450 lines of code and offers a slight performance improvement over the base manual variant.

3.6 Pinned Memory Variant

The pinned memory variant is nearly identical to the base manual memory variant, but uses a different kind of host memory. Normal host memory allocated through `malloc` is not pinned (page-locked). However, memory copying between the host and device requires pinned memory. So, unpinned memory is first copied over to a page-locked area in memory, then transferred to the device. By instead allocating pinned memory, the device has direct access to the memory and can transfer it without assistance from the CPU. This not only increases the speed of memory transfers, it also allows for the possibility of asynchronous execution.

Pinned memory can be allocated directly through CUDA API calls. The `cudaMallocHost` function will allocate page-locked memory, and the `cudaFreeHost` function will deallocate it. With faster memory transfers, this variant performs moderately better than the base manual memory variant, while requiring the same amount of code. The performance increase from unified memory to manual pinned memory was larger than the increase for any other CUDA variants.

3.7 Asynchronous Variant

The asynchronous variant was developed to fully utilize pinned memory by overlapping both data transfers and kernel execution. In every other variant, a kernel/loop has to wait until the previous execution has finished before proceeding. By tiling the grids in the z-direction, multiple tiles can be operated on simultaneously, removing any busy waiting.

CUDA streams allow the developer to queue memory transfers (through `cudaMemcpyAsync`) and kernel invocations (by specifying stream in invocation) asynchronously. Similar to threads on a CPU, multiple streams can run concurrently, allowing for parallel device execution. Before execution, the user can specify the desired number of tiles and streams. Using these values, the block/grid dimensions are calculated. Each stream is given a section of memory to transfer and a range of values in the z-dimension corresponding to the current tile. Once the memory necessary for the kernel to read has been transferred, the kernel execution is queued for the current stream. This process repeats until all tiles have been queued on a stream. Finally, calling `cudaStreamSynchronize` for each stream will block the host code until all operations have completed on each stream. Figure 7 shows the approach to tiling the grids in the z-direction and distributing load across multiple streams.

```
//Create the streams
cudaStream_t streams[NUM_STREAMS];
for (int i = 0; i < NUM_STREAMS; i++) {
    cudaStreamCreate(&streams[i]);
}

//Determine block & grid sizes
int gridZ = domain->nz / NUM_TILES;
dim3 blockSize = dim3(32, 32, 1);
dim3 gridSize = dim3(domain->nx / 32,
                     domain->ny / 32,
                     gridZ);

//Operate on each tile
for (int i = 0; i < NUM_TILES; i++) {
    //Compute z-value range for current tile
    int z1 = i * gridZ;
    int z2 = min(z1 + gridZ - 1, domain->nz - 1);

    //Determine memory copy amounts
    //copySize is the total number of bytes to copy over
    int faceSize = domain->nx * domain->ny;
    int memoryOffset = faceSize * z1;
    int copySize = faceSize * sizeof(double) * (z2 - z1 + 1);

    //Copy grid chunks needed for loop, invoke kernel
    cudaStream_t stream = streams[i % NUM_STREAMS];
    cudaMemcpyAsync(..., cudaMemcpyHostToDevice, stream);
    kernel<<<gridSize, blockSize, 0, stream>>>(z1, z2, ...);

    ...
}

//Synchronize, then destroy the streams
for (int i = 0; i < NUM_STREAMS; i++) {
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}

//Copy result grids back to the host synchronously
cudaMemcpy(..., cudaMemcpyDeviceToHost);
...
```

Figure 7: Tiling grids in the z-direction for asynchronous execution

Chapter 4

Experimental Results

The results of the experiment show that all CUDA variants performed better than the OpenCL variant. Additionally, the line counts of the CUDA variants were far less than the OpenCL variant. The best performing CUDA variant utilizing the unified memory system was the C++ lambda variant, which also had the least lines of code out of any variant. The best performing CUDA variant utilizing manual memory management was the asynchronous variant, which had the highest speedup multiplier of all variants. The largest performance increase among the CUDA variants occurred between the base manual memory variant and the pinned memory variant, which is due to an increase in memory transfer speed between the host and device.

4.1 Methodology

All experiments were performed on a single machine called "Rose" in the University of Arizona Computer Science department. Rose contains the following specifications:

- Intel Core i7-6900K CPU (8 cores, hyperthreading disabled)
- 32 GB of RAM
- Nvidia GeForce GTX 1080 GPU

The project utilizes the following for building each variant into separate executables:

- CMake (version 3.16.0-rc4) for build specification
- g++ (version 5.4.0) for the compilation of normal C/C++ code
- nvcc (version 9.0.176) for the compilation of CUDA code

Additionally, Nvidia's nvprof (version 9.0.176) profiler was used to gather metrics for CUDA memory transfers and kernel timings.

The variants were developed in the same order specified in the introduction. After a new variant was developed and verified for correctness (by comparing its output against the serial variant's output), it was then ran a total of 10 times with grid sizes of 250x250x250, where LowFlow will output the timing metrics desired. These timing metrics were placed in a table and averaged. The speedup multiplier for a variant is calculated by dividing the average runtime of the serial variant (5.768 seconds) by the average runtime of the variant in question. This gives an idea of how much faster a variant is over the serial variant.

Some variants require additional parameters before execution. For example, the OpenMP variant relies on a predefined environment variable that specifies the number of threads to use in an OpenMP environment. Metrics were gathered for 1, 2, 4, and 8 threads for this variant. For the asynchronous variant, the user is able to specify the number of tiles and CUDA streams desired. The number of streams is optimal when it is less than or equal to the number of tiles. If the number of streams is greater than the number of tiles, then there will be some streams that do nothing and add unnecessary overhead. After running this variant with multiple combinations of tiles and streams, it was found that splitting the grid into 3 tiles over 3 streams was the optimal amount, though the delta between combinations was still small.

4.2 Metric Comparisons

Figure 8 below shows a table with the total elapsed runtime, total lines of code, and speedup multiplier for each variant. For the OpenMP variant results, the thread count used is attached to the name. For the asynchronous variant, the metrics shown are for the fastest combination of the number of tiles and streams, which as specified before is 3 tiles and 3 streams. Figure 9 shows a bar chart comparing the speedup multipliers.

Figure 8: Runtime, line count, and speedup multiplier for each variant

Variant	Total Elapsed Runtime (s)	Line Count	Speedup Multiplier
Serial (baseline)	5.768	338	1
OpenMP (1)	8.305	296	0.694
OpenMP (2)	4.635	296	1.244
OpenMP (4)	2.584	296	2.232
OpenMP (8)	1.616	296	3.569
OpenCL	0.839	1017	6.872
Base Unified Memory	0.583	342	9.894
Fused Kernel	0.582	300	9.909
C++ Lambda	0.574	248	10.047
Base Manual Memory	0.621	464	9.293
Extracted Struct Members	0.613	543	9.413
Pinned Memory	0.512	464	11.275
Asynchronous	0.497	560	11.598

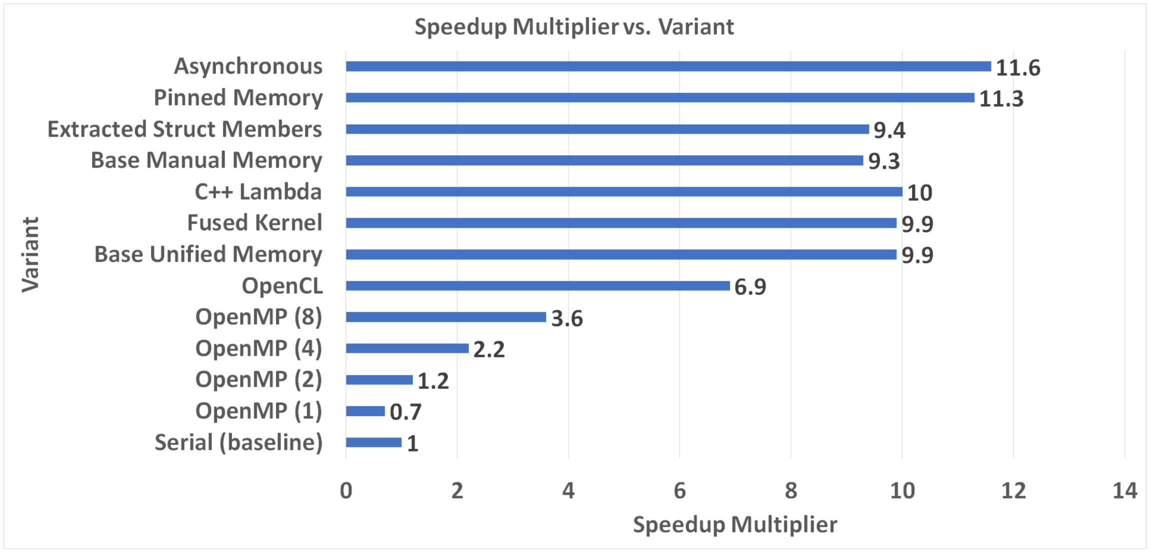

Figure 9: Graph of the speedup multiplier for each variant (longer bars are better). The fastest variant utilizes pinned host memory, manual device memory management, and asynchronous execution.

As shown in the data, OpenMP with 1 thread performs worse than the serial variant. This is due to the additional overhead associated with utilizing OpenMP. Once more than 1 thread is introduced, performance starts to increase. At a maximum of 8 threads, the speedup multiplier stops at 3.6. Adding more threads than available on the CPU would most likely result in a decrease in performance, as threads would begin to compete for CPU resources. After testing OpenMP, tests for OpenCL were ran. With almost twice the maximum speedup of OpenMP, OpenCL highlights the difference between CPU and GPU parallelization. OpenCL however has a much higher line count than any other variant, making it the most difficult to develop.

After OpenCL tests were ran, CUDA development began. The first CUDA variant developed was the base unified memory variant. This variant was very easy to develop, having nearly the same amount of code as the serial variant. As shown, with a 9.9x speedup, even the simplest CUDA variant offers a significant increase in performance over the serial variant.

The fused kernel variant did not perform as initally expected. My initial guess was that fusing the kernels would remove enough overhead to warrant a 10x speedup. Instead, the results show that the extra parameters and additional kernel code balance out the setup and cleanup of multiple kernel invocations.

Surprisingly, the lambda variant performed better than the base unified memory variant. With the addition of the kernel wrapper and the utilization of C++ lambda functions, I expected performance to decrease in favor of simpler code. The lambda function implemented captures all external variables in the enclosing scope by value, which led me to assume that performance would decrease. The increase in performance over the base unified memory variant is likely due to compiler optimizations, such as the possible inlining of smaller lambda functions.

After the lambda variant was tested, all new variants developed use manual memory management. The base manual memory variant performs the worst out of the CUDA variants, which is likely due to requiring 3 memory transfers from the host to the device for each grid (as shown in Figure 6). Even though the third transfer is simply copying over 2 pointers to the device, that memory still needs to be transferred to a page-locked area before moving it onto the device. This additional overhead is the main cause of the performance decrease from unified memory to manual memory.

The extracted struct members variant was developed with the intention of reducing the number of memory transfers for each grid from 3 to 2. While the performance did increase over the base manual memory variant, it is still worse than any other unified memory variant. Additionally, this variant required a rewrite of the kernels that previously referenced a Grid struct to instead reference its members as separate parameters. Some kernels require nearly 20 parameters, which is doubled as a result of replacing a single Grid struct with 2 member variables.

The pinned memory variant was developed to cut out the additional transfer from unpinned to pinned memory when attempting to transfer data to the device.

By directly allocating pinned memory through convenient CUDA API calls, transfers occur much faster, resulting in the large performance jump from memory allocated through `malloc` to memory allocated through `cudaMallocHost`. Pinned memory also allows for asynchronous execution, which was the final step in the development of CUDA variants.

Up until the pinned memory variant, each kernel would wait until the previous kernel had finished its execution, which operated on entire grids. To hide any busy waiting, the overlapping of memory transfers and kernel execution was implemented. The asynchronous variant was the most difficult CUDA variant to develop, as an additional layer of memory management is required from the developer. When tiling a grid in the z-direction, it is important that the exact range of z values pertaining to a tile are operated on, and that the correct amount of memory is transferred over to the device. Additionally, two more parameters must be controlled to ensure maximum performance: the number of tiles to split the grids into and the number of CUDA streams (similar to threads) that must be spawned. As shown, this variant performs slightly better than simply using pinned memory and offers the highest performance increase over the serial variant.

Chapter 5

Conclusion

From the results of testing multiple variants of CUDA code, it is evident that parallelization through CUDA offers a far greater increase in performance than its rival, OpenCL. CUDA development is made easy through an API with functions similar to standard C/C++ code. As a result, transitioning from serial implementations to CUDA driven parallelizations requires minimal changes. Performance can be maximized by manually managing memory, which offers a high level of control over interactions between the host and device. The disadvantage to CUDA code is its restriction to running solely on Nvidia devices. OpenCL code can run on multiple architectures, which is useful when attempting to maintain the performance benefits of parallelism across a wide range of devices. However, if a user's machine contains a Nvidia GPU, parallelization through CUDA should be utilized to grant the highest amount of performance possible.

One way to further validate the performance metrics for each variant is to work with a larger section of extracted code from HydroFrame's codebase. The code currently implemented by the variants is well-suited for parallelization, but could still benefit from testing the framework on a larger scale. Additionally, having the variants run on multiple time steps would give a better insight of how the code performs over a much longer period of time. Despite this, the metrics shown still provide a snapshot of the performance benefits possible through CUDA parallelizations.

References

[1] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5. Technical report, November 2015.

[2] Mark Ebersole. What Is CUDA? *NVIDIA Official Blog*, September 2012.

[3] Khronos Group. OpenCL 2.0 Reference Guide. Technical report, 2015.

[4] NVIDIA. CUDA Toolkit Documentation v10.2.89. Technical report, November 2019.