CIS*4900 Final Report
Parallelizing Nvidia Warp's CPU Mode

Roman Savelyev

April 18, 2023

1 Introduction

Nvidia Warp is a Python framework that allows its users to write kernel-based code in Python. In short, Warp utilizes *just-in-time (JIT)* runtime compilation that ultimately converts Python functions into C++ and CUDA kernel level code[2]. The main purpose of Nvidia Warp was to create a user-friendly interface that is built on the CUDA platform for coders to create high performance simulations and graphics using Nvidia's own GPUs[4]. Although Nvidia Warp is mainly used to write programs that are parallelized on Nvidia GPUs, Warp does offer an option to run its code using the system's CPU – this is typically done when the given system does not have access to an Nvidia GPU or the Nvidia GPU's compute capability is below 3[1]; the maximum number of threads per block on the GPU is determined by this value[1]. The caveat that comes with using the CPU for acceleration in Nvidia Warp is that the code no longer runs on the CUDA platform, and instead gets compiled into a C++ program that runs in serial as opposed to parallel by utilizing only one thread per kernel. Thus, the purpose of this project is to explore how the OpenMP shared memory API could be used to add multi-threading to Warp for executing it on the CPU. This project will identify the potential obstacles, such as shared global state, and investigate options for refactoring / re-designing the code to make it more amenable to auto-parallelization with OpenMP. Similarly, the project will aim to achieve two learning goals: (1) - gain experience in working with parallel computing and optimization and (2) - gain experience solving a real world problem, as opposed to solving problems created for the purpose of teaching a concept.

2 Proposed Method

As mentioned before, Nvidia Warp will use just-in-time compilation that will map supported Python functions during runtime to convert them into C++ or CUDA depending on whether the CPU or GPU was used as the device, respectively. As this process is done during the runtime, there exist multiple functions that will map out and finally generate the code that will later be compiled and ran to complete the program. The proposed method was to edit the generation of the CPU C++ code and implement OpenMP to run the kernel on multiple threads as opposed to one.

3 Overview

3.1.0 Introduction to OpenMP

OpenMP is an API for shared-memory MIMD (Multiple Instruction, Multiple Data) programming[1]. Similar to an API like Pthreads, each thread used by OpenMP has access to all available memory. Unlike other multithreading APIs, including Pthreads, OpenMP was developed to be used at a higher level and allow programmers to incrementally parallelize existing serial code[1]. This refers to a technique of parallelizing existing code as a sequence of changes, parallelizing one loop at a time, minimizing the introduction of bugs into the code.[6] The ease of implementing OpenMP paired with its robustness is one of the main reasons as to why it was considered as part of the proposed solution. Running OpenMP is as easy as setting a #pragma *args* decorator above the code that will be parallelized – with args mainly including the number of threads as well as the type of parallelization, which in the case of Nvidia Warp was for-loop parallelization. For-loop parallelization refers to OpenMP's ability to parallelize a pre-existing for-loop by deciding on its own which specified thread will take each iteration of the for-loop.

3.2.0 Writing Code in Nvidia Warp

The general process for writing code in Warp is indifferent to whether the code is running on the CPU or GPU. The process follows four basic steps:

1. Import Warp – usually renamed by importing Warp as 'wp'
2. Initialize Warp – using wp.init()
3. Write the kernel function – using @wp.kernel and by writing the function below
4. Launch the kernel – using wp.launch(*args*)

The arguments that will be passed into wp.launch(), defined in step 4, play a crucial part in how Warp will treat the code. The 4 required arguments will include:
kernel - This variable will be assigned the function Warp will be using JIT compilation on and should match the function name written below @wp.kernel. If multiple kernels exist, each should have a corresponding launch.[3]
dim - This is the block-size or in other words, the number of threads to launch the kernel on. This value can be an integer or a tuple of ints with a maximum of 4 dimensions. Extra dimensions are added for the purpose of convenience for graphical design, but in general, if *dim* is equal to a tuple of ints, the number of threads will be the product of the integers inside the tuple.[3]

inputs - This is a list of arguments to pass to the function defined in *kernel.*[3]
device - This is the device that Warp will run the code on, and most importantly will define how the code is run and how the JIT compilation works. Typically the two options for this are ‘cuda’ or ‘cpu’.[3]
Since the purpose of this research is to improve the CPU mode in Nvidia Warp, the CPU is the device that will be focused on.

3.2.1 Launching Nvidia Warp Using CPU

When the user runs the code, Nvidia Warp will convert all of the variables and supported functions under the @wp.kernel decorator into C++ values and will store these values in lists to be used for later. The lists are checked to ensure that requested function calls are supported by Warp. The most important step will happen once the user code is fully parsed and converted, and will take place inside the *codegen.py* program within Warp’s library. Depending on the device used, *codegen.py* will use pre-existing templates to generate the header and functions of the new C++ file that will be populated using the previously made lists that store the converted function names, arguments and variables. The C++ file is then compiled and ran. Figure 1 shows visualization of this process: the Python code is converted into C++/CUDA; it is compiled with the shared libraries which include builtin arithmetic functions and ran.

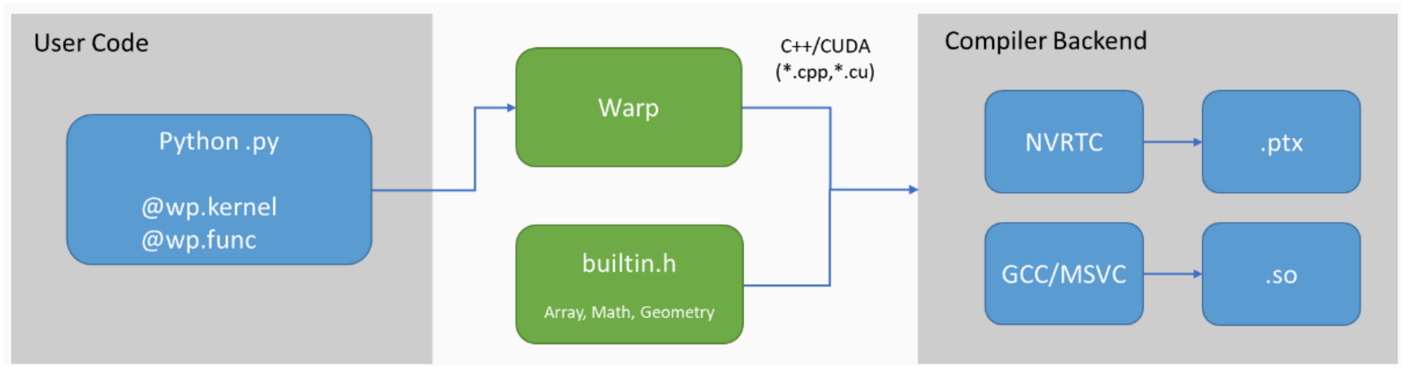

Figure 1: Model of how Nvidia Warp compiles/runs its code[2]

3.2.2 Generated C++ File

As explained in section 3.2.1, when Nvidia Warp launches using the CPU, a C++ file is generated that is a rewritten version of the Python code written by the user. Within this file there exist 4 functions. The functions shown in Figure 2 are responsible for holding the C++ JIT converted code that will be called by the functions in Figure 3. The function calls happen within a for-loop, shown in Figure 4. The for-loop uses a variable *i* as the iterator, which is also assigned to the global variable *s_threadIdx* which is used by other functions in the linked files to get the thread ID of the current running thread, as later shown in Figure 5.

```
void name__cpu_kernel_forward(args)
void name__cpu_kernel_backwards(args)
```

Figure 2: Kernel forward/backward functions that hold the user written C++ JIT compilation output. The file this is found in is given when the code is running, printed to STDOUT with heading 'Kernel Cache:'[5]

```
void name__cpu_forward(args)
void name__cpu_backward(args)
```

Figure 3: Cpu forward/backward functions that include a for-loop thread counter with each iteration making a call to the kernel_forward or kernel_backward functions shown in Figure 2. [5]

```
for (int i=0; i < dim.size; ++i)
    {
        s_threadIdx = i;

        name__cpu_kernel_backward(dim,
            var_input,
            adj_input);
    }
```

Figure 4: The for-loop structure that can be found under the functions in Figure 3 [5]

4 Solution

4.1.0 Overview

As shown in Figure 4, within the generated C++ file, there exists a for-loop that launches the kernel function at each iteration, each iteration being a reference to the thread count. The main idea was to use OpenMP to parallelize this for-loop, reducing the total number of serial iterations from dim to $\frac{dim}{n}$ where dim is the block size defined in section 3.2.0 and n is the number of threads used by OpenMP. The performance and efficiency will be limited to the user's CPU. If $n > Number\ of\ Cores$, then it can be safely assumed that the program will not run efficiently. In the case of CPUs that use hyperthreading, $Max\ Threads = 2 * Number\ of\ Cores$.

4.2.0 Change I

The first major change was the introduction of OpenMP within the Warp library. This set of changes was concerned with the OpenMP parallelization of the for-loop within the generated

C++ file, shown in Figure 4, and the compilation of OpenMP. When this change was introduced and tested on a simple thread counter program, it mimicked parallel behavior by printing thread outputs in the order they started as opposed to the order of the for-loop.

4.2.1 Change II

The reason Change I resulted in only the mimicking of parallelization is because the thread counter *s_threadIdx = i* shown in Figure 4 acts as a critical section – a segment of code that tries to access or modify the variable in a shared access;[1] the variable *s_threadIdx* being a global variable. One way to fix this would be to lock and unlock the mutex at each iteration of the for-loop to allow the safe assignment of *s_threadIdx*. The issue is that even with OpenMP parallelizing the code, the program's performance would still be that of a serial program if not worse due to every thread locking and unlocking, adding to the runtime. Change II had to do with finding a way to get the thread ID without a critical section. One thing worth noting, *s_threadIdx* is used as a return variable only by functions that get the thread ID, for example, wp.tid() shown in Figure 5, which is called from the kernel in the user created Python code. The goal was to find an alternative to a wp.tid() call for if openMP is being used. This was done by passing the thread ID to the functions shown in Figure 2 as an argument, and making a blank function *wp.tid_omp()* that will trick the program into assigning the passed thread ID argument to the variable defined by the user during the JIT compilation process. To get more technical, since *wp.tid_omp()* returns 0, a check was implemented in codegen.py, such that if *wp.tid_omp()* is called, instead of assigning the function to the user defined variable, the program will catch that moment and assign the passed thread ID argument instead. Once this was done, the thread ID was no longer linked to a global variable, and each function had access to its own private copy of the thread ID as opposed to a shared variable that stores it, resulting in the removal of the critical section.

```
inline CUDA_CALLABLE int tid()
{
#ifdef __CUDACC__
    return blockDim.x * blockIdx.x + threadIdx.x;
#else
    return s_threadIdx;
#endif
}
```

Figure 5: The tid() function called as wp.tid() found in warp-main/warp/native/builtin.h as well as warp-main/build/lib/warp/native/builtin.h[5]

4.2.2 Results

The program used to get results is a matrix multiplication calculator that multiplies matrix A of size $x \times y$ and matrix A^T of size $y \times x$. When multiplying a matrix of size $a \times b$ by a matrix of size $c \times d$, the resulting matrix size will be of size $a \times d$. Thus for the purpose of getting more distinctive results while testing each configuration, the x value is the variable being increased. The resulting matrix for each test will be of size $x \times x$. The tests were conducted using an Nvidia GTX 1080 Ti GPU, with 3584 CUDA Cores and a compute capability of 6.x[7], and an Intel i7-6700k CPU, with a total of 4 cores and 8 threads due to its hyperthreading ability.

	Nvidia Warp Configurations				
x,y values	**CPU - wo/ OpenMP**	**CPU - w/ OpenMP - 2 Threads**	**CPU - w/ OpenMP - 4 Threads**	**CPU - w/ OpenMP - 8 Threads**	**GPU (Cuda)**
1000,1000	4.81s	4.41s	3.86s	3.71s	1.50s
2000,1000	10.93s	8.18s	6.88s	6.57s	3.44s
3000,1000	19.27s	13.63s	10.54s	9.61s	6.93s
4000,1000	32.04s	20.90s	15.76s	13.80s	10.08s
5000,1000	46.74s	30.03s	21.66s	18.32s	16.49s
6000,1000	64.38s	39.72s	29.44s	25.24s	22.86s
7000,1000	85.71s	51.92s	36.97s	31.44s	30.64s
8000,1000	110.37s	65.74s	46.46s	38.33s	39.71s
9000,1000	137.99s	82.64s	56.76s	45.42s	49.65s
10000,1000	168.23s	99.11s	67.65s	54.46s	59.96s
	Time is recorded once the program has cached the output to prevent Nvidia Warp's JIT compilation from interfering with the results				

Table 1: Running times of matrix multiplication using different Nvidia Warp configurations

	Parallelized Nvidia Warp Configurations			
x,y values	**CPU - w/ OpenMP - 2 Threads**	**CPU - w/ OpenMP - 4 Threads**	**CPU - w/ OpenMP - 8 Threads**	**GPU (Cuda)**
1000,1000	1.09x	1.25x	1.25x	3.21x
2000,1000	1.34x	1.59x	1.66x	3.18x
3000,1000	1.41x	1.83x	2.01x	2.78x
4000,1000	1.53x	2.03x	2.32x	3.18x
5000,1000	1.56x	2.18x	2.55x	2.83x
6000,1000	1.62x	2.19x	2.55x	2.82x
7000,1000	1.65x	2.32x	2.73x	2.80x
8000,1000	1.68x	2.38x	2.88x	2.78x
9000,1000	1.66x	2.43x	3.04x	2.78x
10000,1000	1.70x	2.49x	3.09x	2.81x

Table 2: A speedup comparison of the different parallelized modes compared to CPU - /wo OpenMP based on the values in Table 1

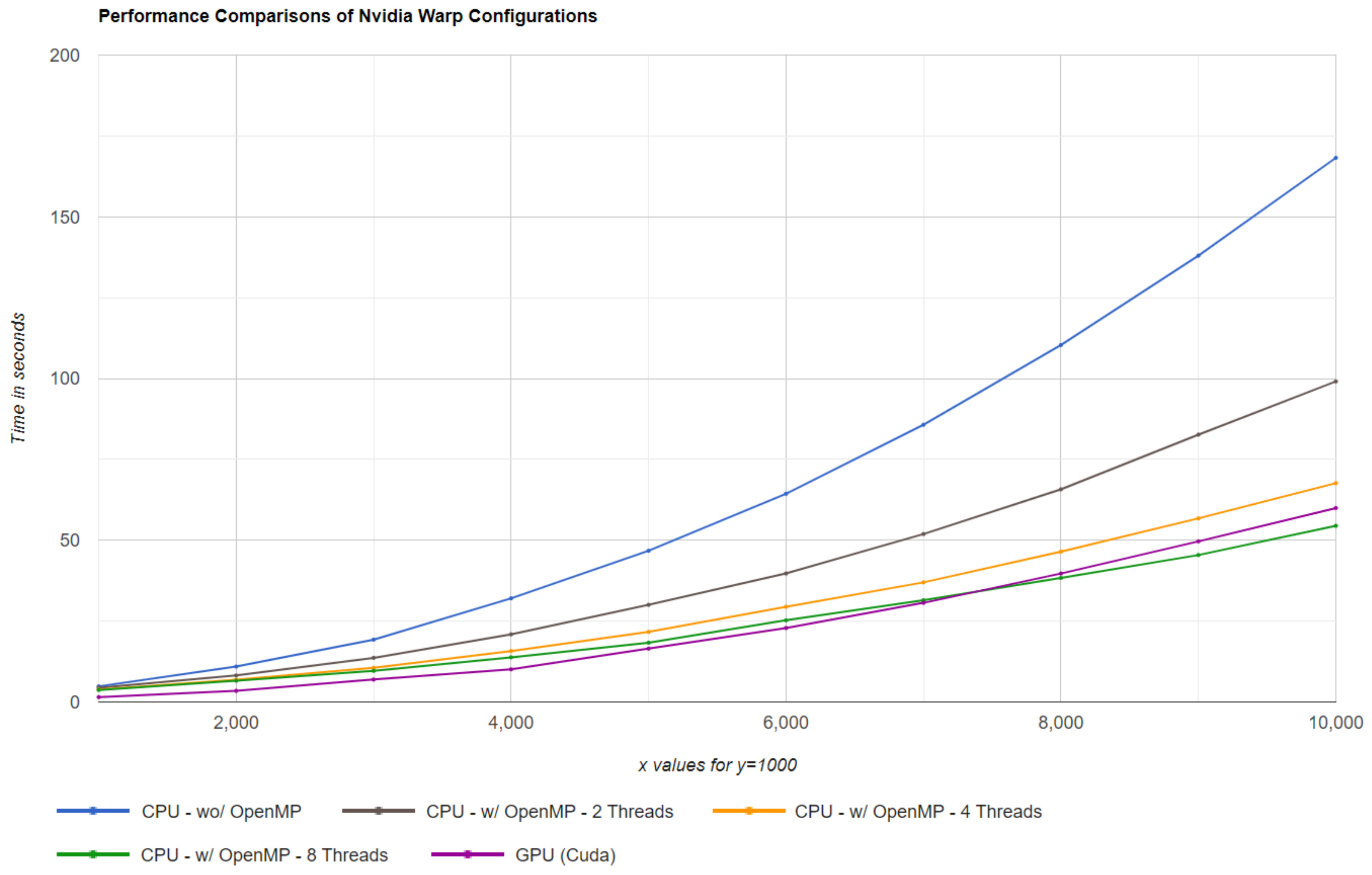

Figure 6: Graph of running times of different Nvidia Warp configurations calculated in Table 1

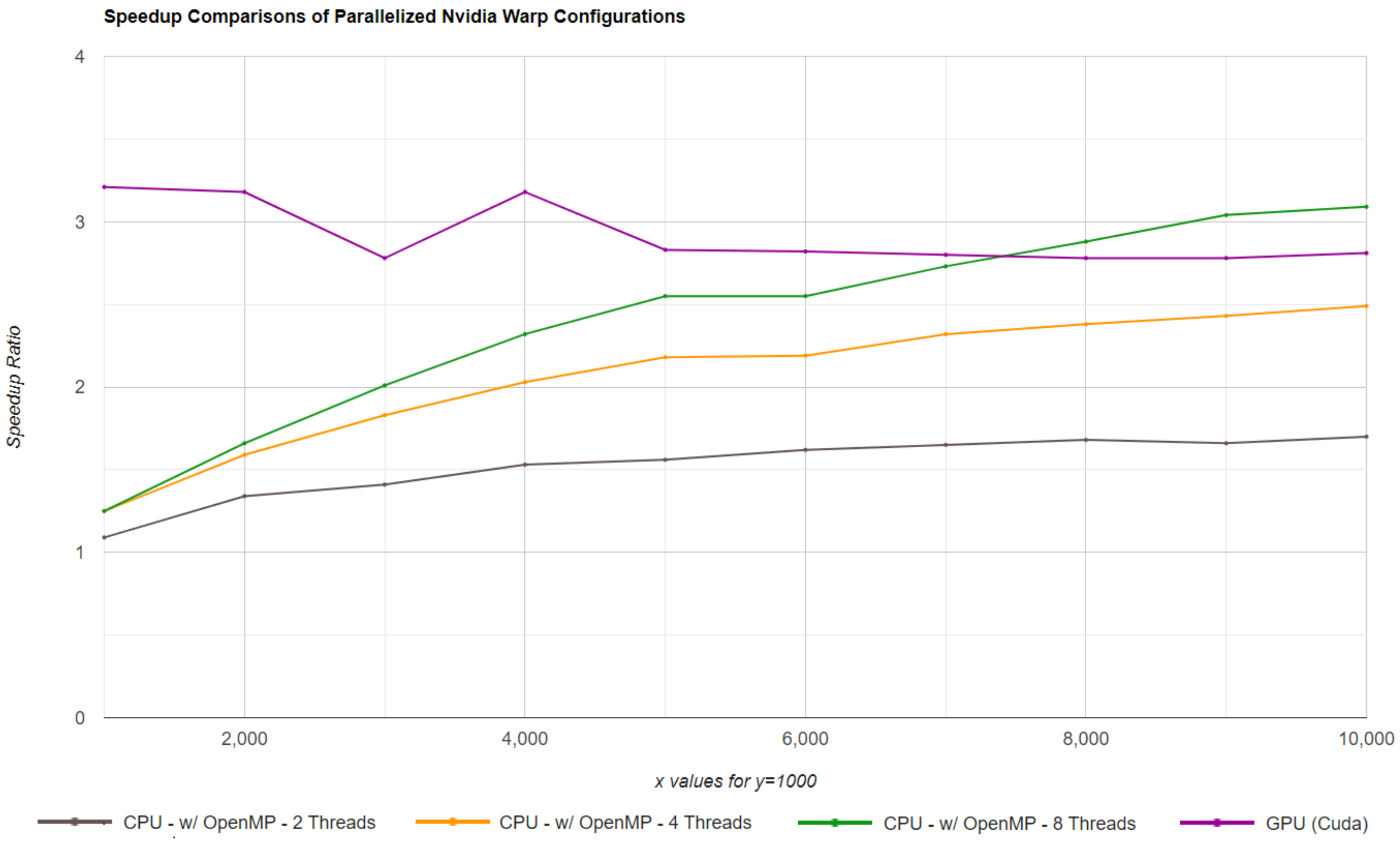

Figure 7: Graph of speedup ratios calculated in Table 2

Table 1 and the graph shown in Figure 6, show the running times of different Nvidia Warp configurations, and portray clear time improvements as the number of cores increases. What becomes interesting, is that somewhere between x = 7000 and x = 8000, *CPU - w/ OpenMP - 8 Threads* becomes more efficient than *GPU (Cuda)*. The same results can be observed in Table 2 and Figure 7 which show the speedup ratios of parallelized Nvidia Warp configurations compared to the base *CPU - /wo OpenMP* configuration, values for which are shown in Table 1. It can be speculated that this has to do with the number of CUDA cores available on the GPU, or as mentioned in section 1, this could be due to the thread limit set by the GPU's compute capability. Similarly this can also be due to the high cost of memory transfer to the GPU. Since no further research was made to support this, these claims are nothing but speculations, and should be subjected to future research. It can also be observed in Figure 7 that the speedups using *CPU - w/ OpenMP - 2 Threads* and *GPU (Cuda)* are starting to straighten out and nearing a slope of 0, indicating a constant speedup, while *CPU - w/ OpenMP - 4 Threads* and *CPU - w/ OpenMP - 8 Threads* is showing the speedup gradually increasing, possibly to a much larger ratio if the *x* value were to be increased. Potentially with an *x* value large enough, even *CPU - w/ OpenMP - 4 Threads* might become more efficient than *GPU (Cuda)*.

5 Conclusion

5.1.0 Conclusion

The purpose of this research was to find and implement a solution for auto-parallelization in Nvidia Warp when using it with the CPU, which was limited to a single-thread usage. This objective was achieved, with strong results to support it. As the majority of research was focused on learning the code as well as implementing fixes to get parallelization to work, there is lots of room left for both future improvement of the new code and further research possibilities. Even with OpenMP built into the code and working, certain functionalities such as barriers are missing. This can lead to problems for when the use case comes up. Similarly, some questions about the performance of the code are left unanswered, such as why did the CPU become more efficient at some point compared to the GPU.

5.2.0 Learning Goals Reflection

One of the learning goals defined in the research project was to gain more experience working with parallel computing and optimization, which was done through working with Nvidia Warp and OpenMP. Nvidia Warp, although created by a large company, provided little documentation of how everything works. This fact makes the experience gained much more significant as it is less based on reading the material on how to do something, and more based on trying different tactics out until something works. The same idea can be applied to the second learning goal which was to solve a real world problem, as opposed to the generic problems given in other classes in order to teach concepts. It can also be added for the second goal that Nvidia wanted to fix this issue themselves and were looking for ways to do it, but did not have the time to dedicate to getting it done.

6 Sources

[1] P. S. Pacheco and M. Malensek, “An Introduction to Parallel Programming Second Edition” Morgan Kauffman, 2022.

[2] “NVIDIA Warp Documentation,” Introduction. [Online]. Available: https://nvidia.github.io/warp/_build/html/modules/introduction.html. [Accessed: 17-Apr-2023].

[3] “NVIDIA Warp Documentation,” Runtime Reference. [Online]. Available: https://nvidia.github.io/warp/_build/html/modules/runtime.html. [Accessed: 17-Apr-2023].

[4] “Nvidia Developer” NVIDIA Warp—Preview Release. [Online]. Available: https://developer.nvidia.com/warp-python. [Accessed: 17-Apr-2023].

[5] NVIDIA, ”NVIDIA/warp” GitHub. [Online]. Available: https://github.com/NVIDIA/warp. [Accessed: 17-Apr-2023].

[6] University of Florida “A Pattern Language for Parallel Programming” Glossary: Parallel Computing. [Online]. Available: https://www.cise.ufl.edu/research/ParallelPatterns/glossary.htm#:~:text=Incremental%20parallelism%20is%20a%20technique,one%20loop%20at%20a%20time. [Accessed: 17-Apr-2023].

[7] “CUDA” CUDA C++ Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities. [Accessed: 17-Apr-2023].