Comparing CUDA to OpenMP Accelerator Support for Graphics Processing Unit

Aman Chopra*
Computer Science Department
New York University
New York, NY, USA
ac8511@nyu.edu

Riju Khatri*
Computer Science Department
New York University
New York, NY, USA
rk3766@nyu.edu

Mohamed Zahran
Computer Science Department
New York University
New York, NY, USA
mzahran@cs.nyu.edu

***Abstract*—In recent years, due to the enormous amount of data, and the advancements in Big Data Applications and Machine Learning, the Graphics Processing Units are heavily used. For computationally heavy workloads, almost all the cloud services heavily rely on Graphics Processing Unit. We know that CUDA is one of the most mature GPU Languages but due to the need for GPU computation, other languages like OpenMP, OpenCL, and OpenACC are extending their support for accelerators. When we have multiple options to do a workload, it is very important to compare them and identify the one best suited for our use case.**

Just the measurement of performance is not enough to identify a better language. Also, there is usually not a single solution that fits best for all the use cases. Hence, in this study, we want to identify which language among CUDA and OpenMP fits better in certain use cases. We want to compare these languages on the basis of programmability, scalability, performance, and overhead.

***Index Terms*—OpenMP; CUDA; Graphics Processing Unit; GPU offload; Accelerators; Performance; Scalability; Programmability; Sieve; Jacobi; Quicksort; nvprof; perf**

I. Introduction

While the Central Processing Unit is designed to do a variety of workloads, the Graphics Processing Unit is designed for specialized workloads. GPUs are usually peripheral devices that are used to accelerate the task. They simultaneously process tasks in parallel which gives massive amounts of speedup. In this context, the CPU is called a host whereas the GPU is called the device. Since GPU is a peripheral device, communication is required from the host to the device and vice versa. Figure 1 shows a high-level overview of how the communication happened between CPUs and GPUs. This process results in communication overhead. We need to first allocate the memory in GPU and do the data transfer.

*All the authors contributed equally to this work.

CUDA is one of the most mature GPU languages. It gives a lot of control to the programmers as to how he wants to execute the tasks. OpenMP on the other hand is a high-level programming language that abstracts these details from the programmer. While this results in ease of programmability but it takes away control from the programmer. The task of parallelizing the code lies in the compiler.

In this paper, the authors have chosen 4 workloads starting from very easy GPU-friendly applications like vector addition to a little complicated parallelized codes of Quick Sort and Jacobi Linear solver. The authors want to compare the two languages OpenMP and CUDA and suggest which approach can be taken given the problem statement and criteria.

The paper has been laid out in the following manner. Section II, the objective section deals with the primary objectives of this paper. Section III of this paper talks about the key studies being already done in this field. Section IV lays down the proposed idea and methodology we have followed to compare these two programming languages. Section V discusses the experiments and results. Sections VI concludes this paper.

II. Objective

In this paper, the authors have taken 4 workloads namely Vector Addition, generating prime numbers using the Sieve of Eratosthenes algorithm, a parallelized version of quicksort, and solving a system of linear equations using the Jacobi method. The aim of this study is to first write efficient code to perform these tasks using CUDA and OpenMP offload to GPU and then compare these two languages on the factors such as efficiency, scalability, quality of code, performance, overhead, and ease of programmability. This will help in having a deeper understanding of the languages' philosophy and

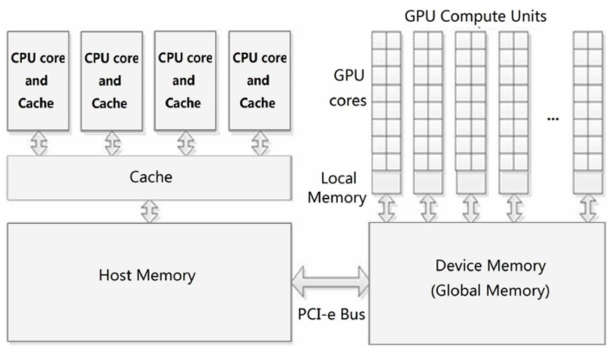

Figure 1. CPU to GPU Communication

one can understand which language could be better to use given a particular use case.

III. Literature Survey

A lot of studies has been done to compare different GPU Programming language. Different studies compare different programs and different aspects of the programming languages. For instance, in [1] the authors have compared the performance of CUDA, OpenMP, and OpenACC on Tesla V100 GPU. They have chosen matrix multiplication as the workload. The study concludes that the OpenMP and OpenACC compilers are able to produce efficient parallel codes for the simple use case whereas the CUDA outperforms them in case of little complex workloads. In [2], the authors have compared the above three languages on three workloads namely Mandelbrot set, N queens problem, and Matrix Multiplication. What I found incomplete about that research was that they only compared these languages based on performance which I believe should not be the only metric. There are a lot of factors that go inside this comparison like scalability, efficiency, control, robustness, etc and we should try to make a thorough comparison.

In [3], the authors did a detailed comparison of OpenMP, and OpenACC. They have compared the portability, performance, data transfer times, kernel execution times, etc of both the languages in many programs like Matrix Multiplication, Image Intensity Transformation, 2D Jacobi Iteration, etc. We have employed a similar approach for OpenMP and CUDA. We believe that our research is holistic as we have not only compared the scalability but also ease of programmability and overhead associated with the languages. We have carefully chosen workloads of different complexity so that we can comment on how the performance of language changes when we change the complexity of the workload.

IV. Proposed Idea and Methodology

In this paper, we will first be talking about some of the CUDA and OpenMP constructs to offload the code to the accelerator. After that, we will be talking about the four workloads we have chosen and how we have tried to parallelize them. We will then be designing many experiments to bring about different comparison aspects like efficiency, scalability, performance, executable size, etc to better understand the intricacies associated with the languages. We will also discuss the challenges we faced while doing this study.

A. Tools

For static analysis, we have used the perf Linux tool [4]. It is a very important tool that gives a lot of hardware and software events like cache-references, LLC-loads, branch-misses which we have used in this study. Another tool that we have used is nvprof [5]. This is part of the CUDA Toolkit to analyze the programs running on the GPU. This can tell us about the communication overhead, the time taken for allocating memory and copying data, the kernel efficiency, etc. Nvprof also gives a lot of metrics and events like warp_execution_efficiency, branch_efficiency, flop_count_sp, etc. Using cuda_profiler_api.h header file, we can profile a specific code segment as well. This is helpful in just profiling the kernel using cudaProfilerStart() and cudaProfilerStop() functions.

B. Language Constructs and functions

For CUDA, these are the important terms and functions:

1) **Kernel**: This is a function executed in the GPU.
2) **cudaMemcpy**: This copies the memory pointed by the source to the destination. This can either be from the host to the device or from the device to the host.
3) **cudaMalloc**: This function allocates bytes of memory in the device and returns a pointer. After the work is done, it is the responsibility of the programmer to free the device memory is cudaFree().

OpenMP was used to write parallel programs for multicore systems. OpenMP 4.5+ has given support so that the parallel code can be run on GPUs. The gcc compilers provide the offloading capabilities as well. For OpenMP, these are the important terms and constructs:

1) **pragma omp target**: Using this construct we can declare a block of code that can be executed in the GPU.

2) **pragma omp target teams**: Using this construct we can create a team of threads. The team has 1 thread by default. This in our understanding is similar to warp in GPU.
3) **distribute**: This construct assigns the loop iterations to teams.
4) **parallel for**: This construct activates the threads at each team and distributes work among them.
5) **map**: This is a very important construct that helps in data movement between the host and the device and memory allocation. We have the to clause for the data movement to the device. Similarly, we have the from and the tofrom clause.
6) Using pragma omp target enter data map, we can keep the data in the device and update our variables. On exit, we can get the data back using pragma omp target exit data map(from: b). This helps in reducing the to and fro data movement and also reduces the communication overhead.

C. *Workloads*

We have chosen the workloads of varying time and programming complexities to better analyze the performance of both the languages. For every workload, we have provided three files. One is for the sequential version and the other two are written in CUDA and OpenMP offload to GPU.

1) ***Vector Addition:*** We start with a very simple program of vector addition. The data, in this case, is a value n. n denotes the size of the vector. We create two input vectors of size n and fill them with random integer values. Our goal is to calculate a sum vector where each element is the sum of the corresponding index elements of the two input vectors.

How to parallelize: The parallelization for this problem was very intuitive. Since each sum subproblem is independent, we can calculate the sum of all indexes parallelly to get the answer. Figure 2 shows the basic process.

2) ***Sieve of Eratosthenes:*** Our second program is to print all the prime numbers from 2 to n, where n is the input from the user. In order to do so, we have used the Sieve of Eratosthenes algorithm and have tried to parallelize it. In this algorithm, we create an array of size n and initially assume that all the numbers are prime. We then start taking numbers one by one and mark all the factors of that number as non prime. In the next iteration, we take the next prime number and repeat the same process. After repeating this process till sqrt(n), we get our final array.

How to parallelize: The parallelization for this problem involved a bit of thinking. The host part took one number and the GPU marked all the factors of that number as non prime. All the threads are working on the same array but on their indexes. Once this process is complete, we copy the device array back to the host array.

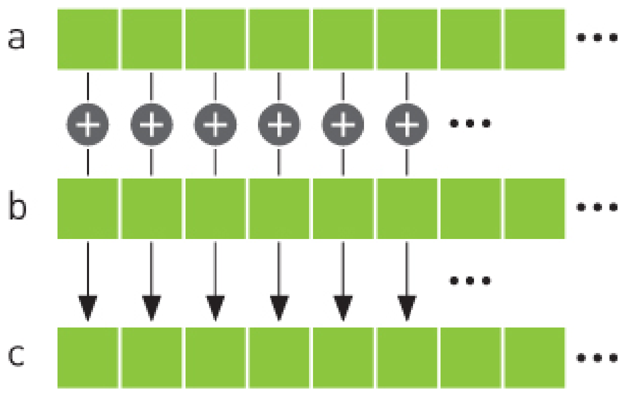

Figure 2. Parallel vector addition

3) ***Quicksort:*** This is a very well-known algorithm to sort an array of numbers. We have tried to parallelize it. The way the sequential quicksort works is that first we select a partition element and try to put all the elements smaller than the pivot element to the left of it and all the elements larger than the pivot element to the right of it. We then repeat this process for both the right and the left subarrays. We keep on doing this until the entire array is sorted. We have created a gen_examples.c file that can be used to generate an input file of n random numbers that needs to be sorted.

How to parallelize: We follow the initial steps as written above and parallelly use the process for the two subarrays. We have used dynamic parallelism in the case of CUDA where the kernel recursively calls itself until a MAX_DEPTH. Dynamic parallelism enjoys many advantages. Firstly, with dynamic parallelism, additional parallelism can be exposed to the GPUs hardware schedulers and load balancers dynamically, adapting in response to data-driven decisions or workloads. Secondly, algorithms and programming patterns that had previously required modifications to eliminate recursion, an irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism can be more transparently expressed. Besides, important benefits when new work is invoked within an executing GPU program include removing the burden on the programmer to marshal and transfer the data on which to operate [6]. Figure 3 shows Dynamic Parallelism.

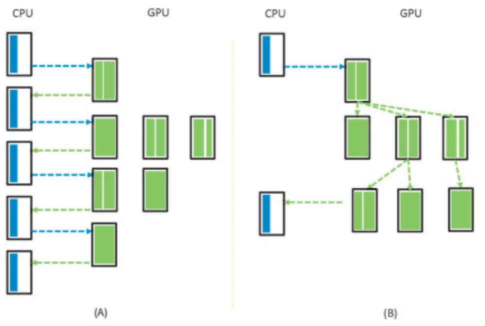

Figure 3. Dynamic Parallelism

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n)$$

$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n)$$

$$\vdots$$

$$x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1})$$

Figure 4. Jacobi Linear Solver

*4) **Jacobi Linear Solver:*** : This is a method to solve a system of linear equations. We have created an executable that generates N equations in the format required by our program. The file contains the number of unknowns and the absolute relative error after which we stop our iteration. It contains the initial value of the unknowns and the coefficients of the equation. We start with a guess and keep on updating the coefficients until the equation's results are within the absolute error. Figure 4 shows how the coefficients can be calculated.

$$x_i = \frac{c_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j}{a_{ii}}, i = 1, 2, .., n \qquad (1)$$

The absolute relative error is given by:

$$|\varepsilon_a|_i = \left| \frac{x_i^{new} - x_i^{old}}{x_i^{new}} \right| * 100 \qquad (2)$$

How to parallelize: We can parallelly calculate the values of x_1, x_2, etc for a particular iteration. After that, we can copy the new values to the old array and repeat the iteration until we get a result less than the required array. If we talk about it in computational terms, it is a relatively more computationally intensive program.

Table I
MACHINE SPECIFICATIONS

Machine Configuration	
Specification	**Value**
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	24
Thread(s) per core	1
CPU MHz	1271.972
BogoMIPS	5000.17
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	30720K

Table II
GPU SPECIFICATIONS

GPU Configuration	
Specification	**Value**
Model	NVIDIA GeForce GTX TITAN Black
Memory	6GB
Bandwidth	288.4 GB/s
Boost Clock	876 MHz
Memory Clock	1502 MHz
Bus Interface	PCIe 3.0*16
Process size	28nm

Table III
VECTOR ADDITION TIMES AND SPEEDUP

N	**Serial Time(s)**	**CUDA Time(s)**	**OpenMP Time(s)**	**CUDA Speedup**	**OpenMP Speedup**
1000	0.003	0.019	0.017	0.15	0.18
10000	0.004	0.024	0.021	0.17	0.20
1000000	0.061	0.271	0.154	0.22	0.40
10000000	0.523	0.398	0.276	1.31	1.89
100000000	3.276	2.272	1.354	1.44	2.42

V. EXPERIMENTS

We have performed all our experiments on CUDA1 machine. Table I contains the CPU Configuration and Table II contains the GPU Configuration.

A. *Vector Addition*

Table III shows the execution times and speedup of CUDA and OpenMP versions. We can see that since the problem is very simple, OpenMP outperforms CUDA in this case. Table IV shows the Last Level Cache loads which give an indication of memory usage. We see that LLC loads for CUDA are way more than that of OpenMP.

Table IV
VECTOR ADDITION LLC LOADS

N	CUDA LLC Loads	OpenMP LLC Loads
1000	688329	14557
10000	768176	15627
1000000	976943	43270
10000000	2088921	383568

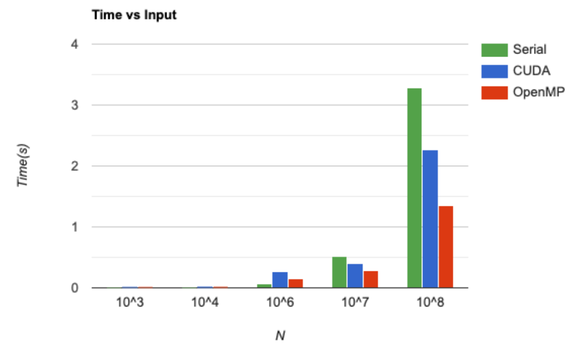

Figure 5. Vector Addition times

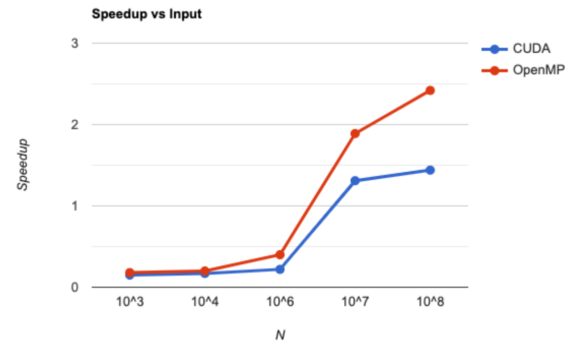

Figure 6. Vector Addition speedup

Figure 5 and 6 shows the timing and speedup graphs.

B. Sieve of Eratosthenes

Table V shows the execution times and speedup of CUDA and OpenMP versions. We can see that since the problem is not very simple, CUDA eventually outperforms OpenMP.

Figure 7 shows the timing graph. Since the problem is a bit complex, CUDA eventually outperforms OpenMP even though OpenMP shows better performance for smaller inputs.

Table V
SIEVE OF ERATOSTHENES TIMES AND SPEEDUP

N	Serial Time(s)	CUDA Time(s)	OpenMP Time(s)	CUDA Speedup	OpenMP Speedup
10000	0.004	0.152	0.019	0.02	0.21
100000	0.014	0.186	0.035	0.07	0.40
1000000	0.061	0.271	0.064	0.22	0.95
10000000	0.436	0.359	0.276	1.21	1.57
100000000	5.342	1.243	1.652	4.29	3.23

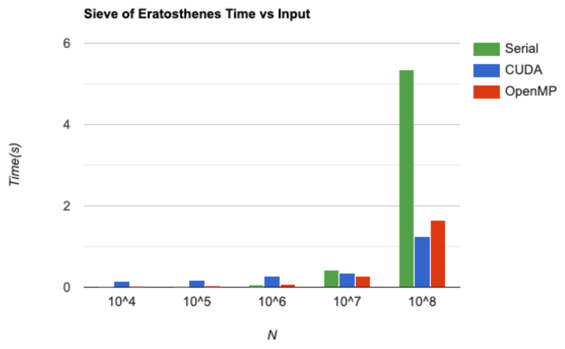

Figure 7. Sieve of Eratosthenes times

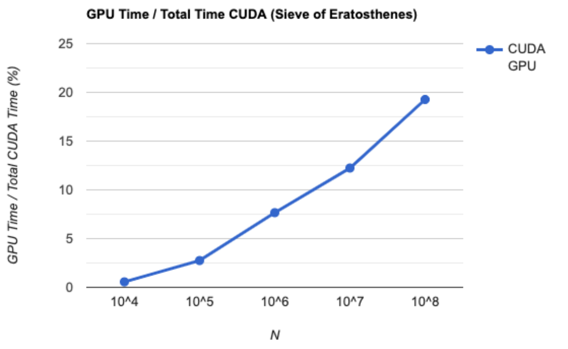

Figure 8. Sieve of Eratosthenes overhead

We have also used nvprof to show the communication overhead. Using nvprof and cuda_profiler_api.h we can just profile the kernel and see how much time against the total time, the kernel is taking. Our results show that there is a huge communication overhead as just the kernel timing was very small compared to the total execution time in the case of CUDA. Figure 8 shows the GPU Time/ Total time for different input sizes N in the case of CUDA.

Table VI
QUICK SORT TIMES AND SPEEDUP

N	Serial Time(s)	CUDA Time(s)	OpenMP Time(s)	CUDA Speedup	OpenMP Speedup
10000	0.042	0.082	0.092	0.51	0.45
100000	0.160	0.182	0.325	0.88	0.49
1000000	0.729	0.344	0.744	2.11	0.97
10000000	2.763	0.729	1.354	3.79	2.04
100000000	12.342	1.472	3.523	8.38	3.50

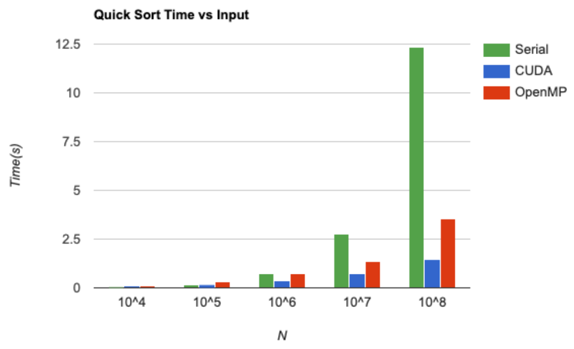

Figure 9. Quicksort times

C. *Quicksort*

Table VI shows the execution times and speedup of CUDA and OpenMP versions. We can see that since the problem is not very simple, CUDA outperforms OpenMP.

In Quicksort, we wanted to know how our program behaves if the input suits the program and when it does not suit the program. For instance, in the case of quicksort, we ran our code with two different sets of inputs. In the first case, we gave the input which is the best case for quicksort. We created cases when the partitions are as evenly balanced as possible. In the second case, we gave a bad input in which the pivot was likely to be either the smallest or the largest element in the list. It is worth mentioning that the results of parallel languages were similar in both cases which shows us that in many scenarios the sequential input norms are not followed in the parallel paradigms. Figure 9 shows the timing graph for quicksort.

D. *Jacobi Linear Solver*

Table VII shows the execution times and speedup of CUDA and OpenMP versions. We can see that since the problem is not very simple, CUDA outperforms OpenMP. Table VIII shows the branch-misses calculated using perf for both languages. It is clear that the branch-misses in CUDA are way more than that of OpenMP.

Table VII
JACOBI LINEAR SOLVER TIMES AND SPEEDUP

N	Serial Time(s)	CUDA Time(s)	OpenMP Time(s)	CUDA Speedup	OpenMP Speedup
10	0.008	0.064	0.198	0.12	0.04
100	0.014	0.105	0.257	0.13	0.05
1000	0.358	0.271	0.372	1.32	0.96
10000	27.35	4.76	7.854	5.75	3.48

Table VIII
JACOBI LINEAR SOLVER BRANCH MISSES

N	CUDA Branch misses	OpenMP Branch misses
10	1220686	401826
100	1723472	562241
1000	2778542	572182
10000	3352423	635373

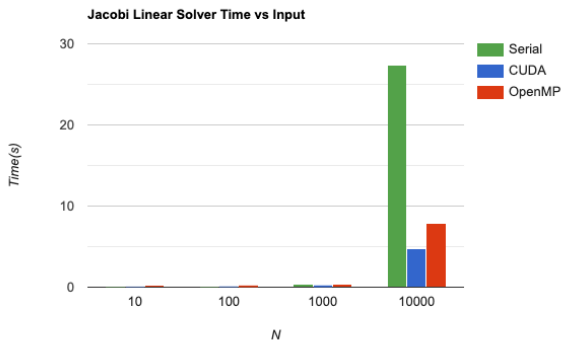

Figure 10. Jacobi Linear Solver times

Figure 10 and 11 shows the timing and speedup graphs. We can see that, unlike Vector Addition where the problem was not at all complex, CUDA in this case shows more speedup than OpenMP.

VI. RESULTS:

We will be comparing our results on different aspects like Ease of Programmability, overhead, performance, portability, etc.

A. *Ease of Programmability*

We discussed that OpenMP is a high-level language. It acts as a wrapper that helps easily write GPU code which the compiler then compiles. It is hence very easy to code because of the level of abstraction it provides. We need not know the intricacies and how the code is

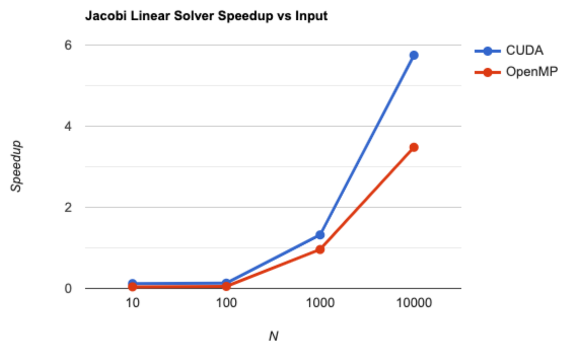

Figure 11. Jacobi Linear Solver speedup

being executed in GPU. A basic level of knowledge of constructs and OpenMP programming is enough. CUDA on the other hand gives more control to the programmer which at the same time means that the programmer is prone to make mistakes. We have to manually allocate memory, copy data, and write kernel code which is not very easy to code. Hence, from the ease of Programmability aspect, OpenMP is the clear winner.

B. *Overhead*

The challenge we faced was that we could not use nvprof to profile the OpenMP offload program. But from our analysis, of the Sieve of Eratosthenes and Jacobi Linear Solver and the results we got from perf, it is clear that there is some overhead associated with CUDA. The resource usage of memory (cache loads) was more compared to that of OpenMP. The communication overhead for CUDA was high as we have seen in Figure 9. With Nvprof too, we saw that a significant amount of time was taken by data movement and device memory allocation. It is clear that CUDA is not worth it for simple small inputs as OpenMP outperforms it.

C. *Performance*

From our analysis of different problems, it is clear that OpenMP performs better with smaller non-complex inputs whereas CUDA performs better with more complex bigger inputs. hence, CUDA is more **scalable** and performs better for huge complex workloads which have significant real-world applications.

E. *Portability*

CUDA is compiled by the Nvidia compiler which is the most important GPU compiler, hence the question of

D. *Executable size*

From Table IX, it is clear that the executable size of CUDA is extremely high compared to that of OpenMP. It also takes relatively more time to compile.

Table IX
SIZE OF EXECUTABLE

Size of executable in bytes		
Problem	**CUDA**	**OpenMP**
Vector Addition	465410	4630
Sieve of Eratosthenes	465477	4310
Quick Sort	1316977	4528
Jacobi Linear Solver	469097	7392

portability does not arise here. CUDA will work well. For OpenMP, Compilers from Cray, Intel, and GCC version 9.1 have support for OpenMP 5.0. However, the support is for a limited set of features. GCC and Cray compilers support only C/C++ base languages and the Intel compiler supports Fortran as well.

VII. Conclusion and Future Work

We have analyzed and compared OpenMP and CUDA in many aspects. We want to conclude that the workload and requirements decide which language is better. There is no clear winner. While OpenMP performs better with small as well as non-complex inputs, CUDA is more scalable. While OpenMP has less overhead, lesser executable size, and is easy to program, CUDA gives more control to the programmers and is one of the most mature GPU Programming languages. There were some challenges we faced like the implementation of Dynamic parallelism for Quicksort and not able to run nvprof for OpenMP. In the future, we want to compare more languages with CUDA and also want to use more profilers to solidify our results.

References

[1] M. Khalilov and A. Timoveev, "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA v100 GPU," *Journal of Physics: Conference Series*, vol. 1740, no. 1, p. 012056, jan 2021. [Online]. Available: https://doi.org/10.1088/1742-6596/1740/1/012056

[2] L. Cleverson, D. Ledur, C. Zeve, and D. Anjos, "Comparative analysis of openacc, openmp and cuda using sequential and parallel algorithms," 08 2013.

[3] R. Usha, P. Pandey, and N. Mangala, "A comprehensive comparison and analysis of openacc and openmp 4.5 for nvidia gpus," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6.

[4] perf: Linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[5] Cuda toolkit documentation. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[6] Dynamic parallelism. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/dynamic-parallelism