Rakrish Dhakal, Sathwik Yanamaddi, Pranav Sivaraman
Dec 12, 2023
CMSC 714: High-Performance Computing

Evaluating OpenMP 4.5 Support on Compilers

1. Abstract

OpenMP is a directive-based API used for parallelizing applications on CPUs. With the release of version OpenMP 4.0 (published in 2013), the API has evolved to let developers offload certain compute-intensive tasks to the GPU. This report evaluates the performance of various compilers in implementing the OpenMP 4.5 API. Through experiments on BabelStream, a memory-bound application, and miniBUDE, a compute-bound application — across the GCC, Clang, NVHPC, and CCE (Cray) compilers, we aim to assess their OpenMP GPU support, uncover performance disparities, and identify areas for improvement.

2. Background

OpenMP (Open Multi-Processing) [5] is a commonly used API for parallel programming using a shared memory model. Developers use pragma directives to specify parallel regions in the code which guide the compiler in generating parallel code. OpenMP handles thread creation and synchronization automatically, distributing the workload among threads and managing their execution.

This ease of usage allows developers to harness the power of multi-core processors for improved performance, thus making it a widely adopted and standardized approach for developing parallel applications.

2.1. OpenMP GPU Support

GPUs have become significantly better at certain compute-heavy tasks than CPUs. However, effectively programming a GPU often requires extensive knowledge of its hardware components and experience using its Instruction Set Architecture. How can programmers harness the power of GPUs without encountering this hurdle?

In 2013, the OpenMP API was extended to support GPU utilization, introducing new annotations that allow programmers to mark code regions that should be offloaded to the GPU [6]. Since then, the OpenMP standard has been updated to improve GPU offloading support. With new, yet simple, directives like *“target”*, the beauty of OpenMP’s GPU support lies in its abstraction.

2.2. Comparison with Other GPU Programming Models

OpenMP allows us to specify parallel for loops and code regions using compiler directives, abstracting the complexity of thread-level programming. There are several alternatives to this style of programming, ranging from low-level, hardware-specific models to higher-level, portable ones.

CUDA [1] is a parallel programming API, and platform, developed by NVIDIA for their GPU systems. Rather than being directive-based, CUDA is a language extension to C, C++, and Fortran, allowing programmers to offload compute-intensive tasks to the GPU by means of new keywords. Since it is hardware-specific, CUDA can only be reused across NVIDIA GPUs, and is therefore not portable to all architectures.

Similarly, **HIP** [2], developed by AMD, is a runtime and language extension of C++. HIP is more portable than CUDA, as it can be used to program both NVIDIA and AMD GPUs. In practice, HIP is more suitable for AMD GPUs, as it requires manual coding and performance tuning to perform as well as a native CUDA implementation on NVIDIA devices.

Finally, **SYCL** [3] is a higher-level programming model with respect to CUDA and HIP. It provides high-level abstractions, as opposed to providing lower-level language extensions. Its primary goal is to provide "heterogeneous" compute by letting developers code using its abstractions, and taking care of compiling kernel code to run on different GPUs. Today, it is most commonly used for Intel GPUs.

Compared to other GPU programming models, the OpenMP model is portable, as programmers can specify high-level compiler directives not tied to any particular hardware. OpenMP also has a much lower learning curve, as it does not introduce any language extensions or higher-level abstractions – letting programmers rapidly offload code regions to GPUs and accelerators using simple directives. Due to these advantages, OpenMP is favorable for many situations over other programming models.

2.3. Motivation

Unlike the other programming models, which are tied to specific runtime implementations, the performance of OpenMP depends on the choice of compiler used to compile the OpenMP-annotated code. As such, it is crucial to determine which compilers provide the best performance for OpenMP GPU offloading, providing the motivation for this project.

3. Experimental Setup

3.1. Applications

To sufficiently benchmark the performance of OpenMP implementations in different compilers, we compile and execute two applications.

BabelStream is an application intended to "measure memory transfer rates to/from global device memory on GPUs" [7]. It is a GPU port of a similar memory-bound kernel developed for CPUs, that instead allocates arrays on the heap, following the best practice for parallel programming. BabelStream has been implemented on a variety of programming models, including the ones mentioned above (CUDA, HIP, and SYCL).

miniBUDE is a molecular dynamics program that runs energy simulations against a protein for a configurable number of iterations [8]. It has also been implemented across a variety of

programming models, and provides options for running benchmarks. Unlike BabelStream, miniBUDE is a *compute-bound* program, meaning that it measures the running time of the kernel, rather than measuring the memory throughput.

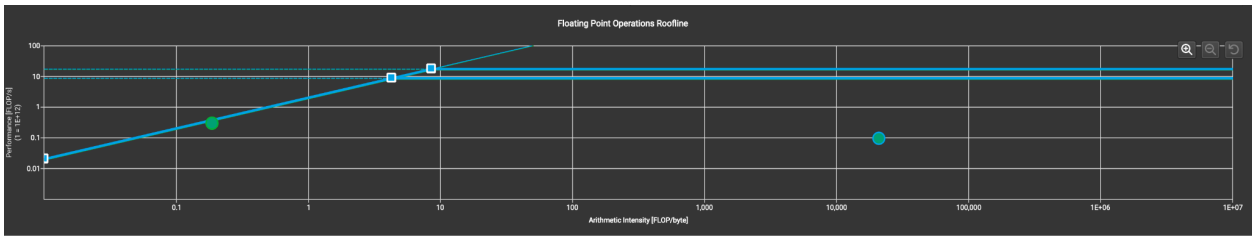

Figure 1. Roofline Plot of BabelStream (Dot Kernel) and miniBUDE

Looking at the roofline plot, we can confirm that BabelStream (dot outlined in green) is a memory-bound application since it lies to the left of the ridge point. For miniBUDE (dot outlined in blue) its kernel lies to the right of the ridge point, indicating that it is a compute-bound application.

Both of these projects are developed and maintained at the University of Bristol, and have been used for benchmarking runtimes and programming models in a number of publications [7, 8]. In this project, we compile the OpenMP implementation of BabelStream and miniBUDE using different compilers (see below). Subsequently, we run each of these compiled programs three times, and compare the average memory throughput, as well as the average kernel time, that each compiled program achieves. By comparing these numbers for each compiled program, we can understand which OpenMP compiler produces the best-performing GPU code, and assess the strengths and weaknesses of each compiler.

3.2. Machine

We run our experiments on *Perlmutter*, a Cray FX supercomputer at the National Energy Research Scientific Computing Center (NERSC). Active since 2021, Perlmutter (officially NERSC-9) is a collaboration of HPE, NVIDIA, and AMD, and has since been dubbed the “world’s fastest AI supercomputer.” [4]

Perlmutter consists of 3,072 CPU-only nodes, each with 2 AMD EPYC 7763 CPUs, as well as 1,536 GPU-accelerated nodes, each with 4 NVIDIA A100 GPUs and 1 AMD EPYC 7763. It features a standard 3-hop dragonfly network, as well as 35PB of flash disk space, augmented with 16 metadata servers and 274 I/O servers [9].

In our experiment, we benchmark the performance of various OpenMP compilers in offloading compute-intensive tasks to one GPU. We measure performance in terms of kernel time and memory transfer speed. Therefore, our experimental setup runs on a single GPU-accelerated node, running on an NVIDIA A100 GPU.

3.3. Compilers

For our experiment, we have chosen to benchmark four OpenMP compilers that are mature, and support both NVIDIA and AMD GPUs.

GCC [10] is an open source compiler that supports multiple languages. It is a general-purpose compiler, and is used across a variety of applications and domains.

Clang [11], like GCC, is also open-source. However, Clang is built on LLVM, a collection of modular compiler and toolchain technologies. As such, Clang has been known to be faster, and use less memory, than GCC. Clang is also a general-purpose compiler.

In addition to Clang, the ***NVHPC*** [12] and ***CCE*** [13] compilers are also built on LLVM. However, these compilers are more domain-specific. NVHPC was developed by NVIDIA, specifically for high-performance computing on NVIDIA devices, whereas CCE was developed by Cray for their supercomputing systems. These are both optimized specifically for their target hardware. Since Perlmutter is a Cray supercomputer that uses NVIDIA GPUs, it will be interesting to see which of these compilers perform better.

4. Results

To evaluate the GPU support of each compiler, we compare the performance of the program that it generates, to the CUDA implementation of each application. Since CUDA is low-level and native to NVIDIA GPUs, it provides a good baseline for comparison. This baseline is used to assess how close each OpenMP compiler gets to the native implementation.

4.1. BabelStream

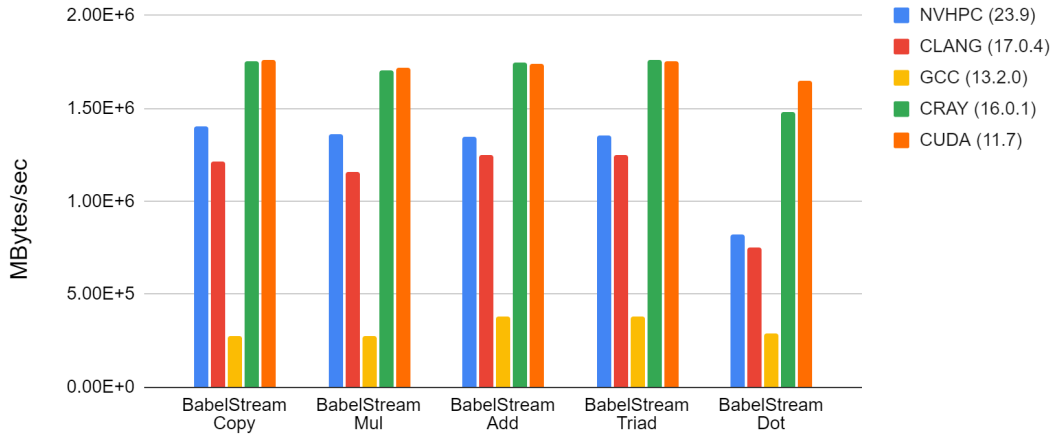

Figure 2. BabelStream Memory Speed Benchmark (Higher is Better)

For BabelStream, we found CRAY outperforms the other compilers and matches CUDA's performance for most benchmarks. CLANG performs slightly worse than NVHPC while GCC performs significantly worse than all other compilers.

4.2. miniBUDE

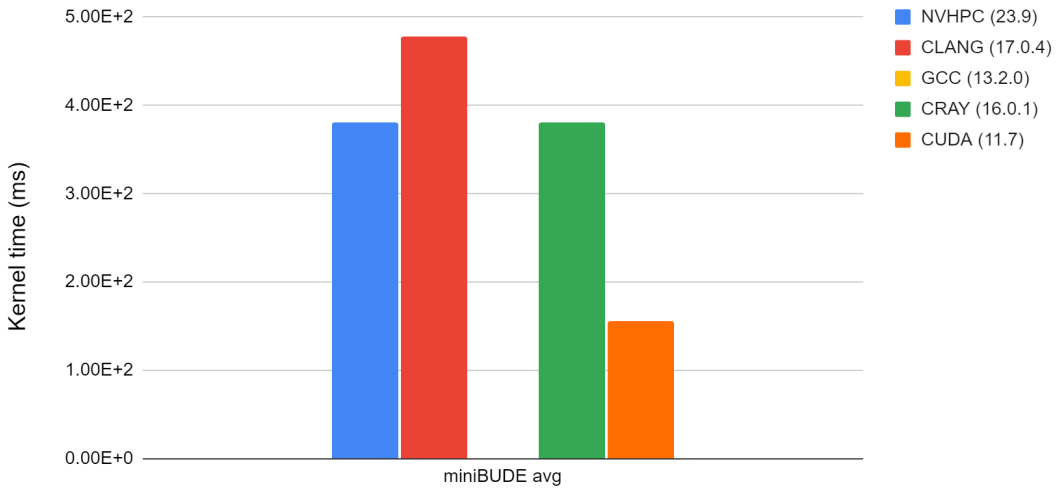

Figure 3. miniBUDE Kernel Time Benchmark (Lower is Better)

For miniBUDE, we found that CUDA significantly outperformed all OpenMP implementations. CLANG performed the worst while NVHPC and CRAY were about the same. GCC did not produce the correct results.

Now, let's look at some interesting cases extracted from these results and investigate how these implementations differ.

5. Analysis

5.1. GCC's Poor Performance

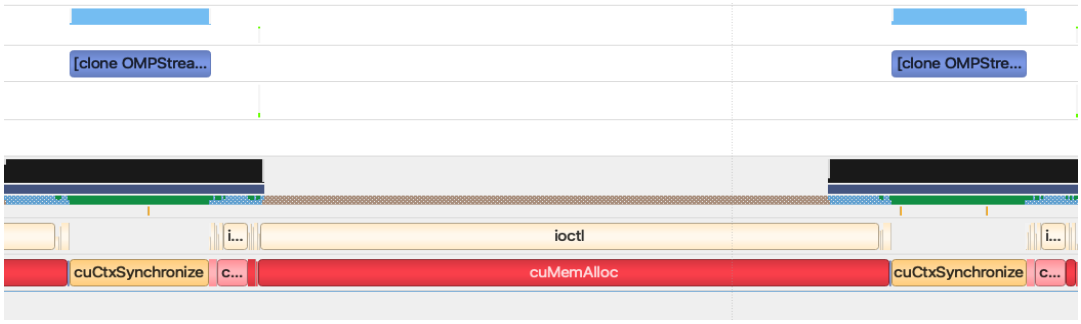

Figure 4. Timeline visualization of BabelStream-GCC (NVIDIA Nsight)

To understand why GCC performs significantly worse than other compilers, we captured the event trace of its BabelStream implementation using NVIDIA's Nsight Systems tool [14]. Looking at the trace, GCC's OpenMP implementation for NVIDIA GPUs seems to perform memory allocations and frees before and after each kernel launch. This behavior is not present in other implementations. Memory allocations are considered costly and ideally should not be performed after every kernel launch.

This issue affects the results in miniBUDE as the main kernel (fasten) is executed multiple times before the final result is sent back to the CPU. Frees and allocations of data between kernel launches lead to the loss of previously computed results, thus, giving us an incorrect output.

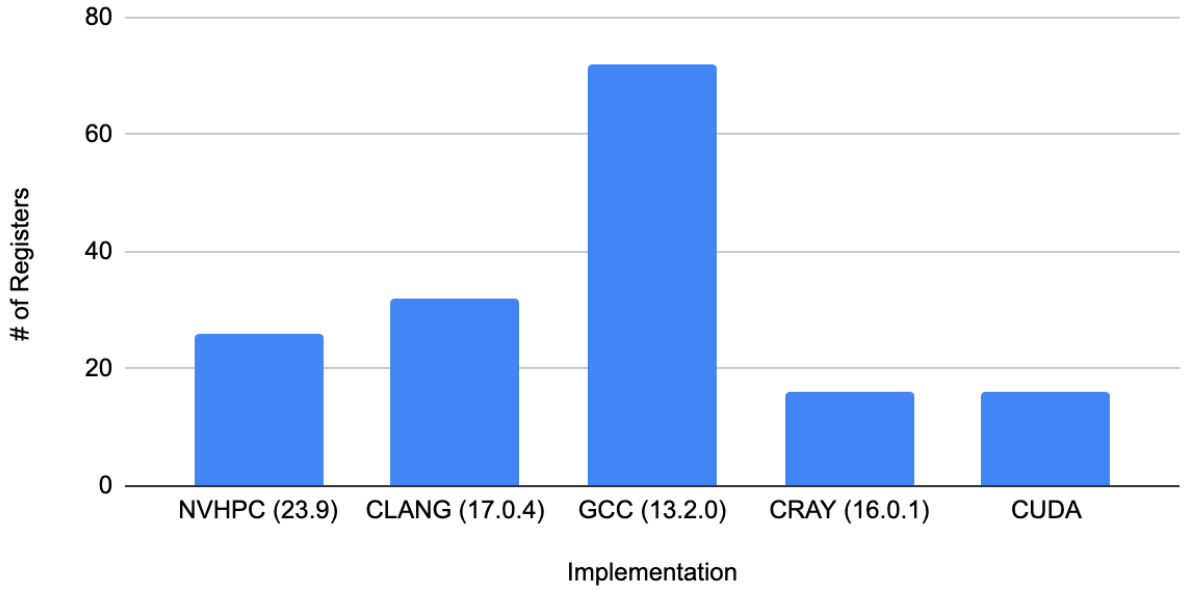

Figure 5. Registers Per Thread, BabelStream Copy Kernel

Examining the GCC kernel profiles reveals a consistent allocation of 72 registers per thread. This high amount limits how many threads can run in parallel on the GPU. Since BabelStream is a memory bound application, threads will often be waiting to fetch from global memory. An SM can hide this latency by scheduling other threads to execute. Other implementations of the simple copy kernel have less than 32 registers allocated per thread which allow for more parallelism and latency hiding.

We investigated the cause for this increased register usage by looking at the assembly code (SASS). GCC's implementation introduces numerous barriers and synchronizations in the SASS. In addition we noticed that it's performing the copy in shared memory rather than in global memory. This is obviously not ideal since there is no data reuse warranted to use shared memory and using it limits the number of threads that can be concurrently executing.

```
LDS.64 R24, [R17]
STS.64 [R17], R24
```

GCC SASS for Copy Kernel
LD (Load Instruction) ST (Store Instruction)

5.2. Why Cray Performs Well

Looking at the earlier registers per thread chart, we can see that the CRAY compiler allocates the same amount of registers per thread as CUDA. Having less registers per thread can allow the GPU to launch more threads and achieve more parallelism.

Looking at the profiles, there are no barriers present in the Cray implementation so threads are not waiting. Additionally, Cray is the manufacturer of the supercomputer so it makes sense to expect good performance, as the hardware is likely optimized for the style of code generated by the Cray compiler.

5.3. NVHPC and Clang's Middling Performance

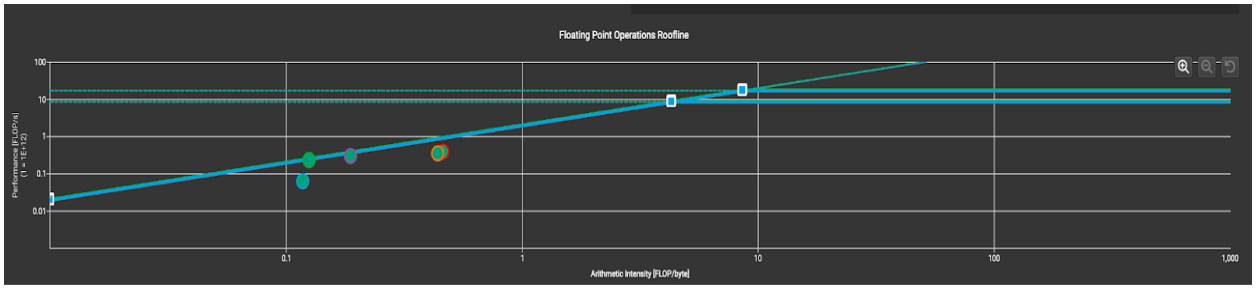

Figure 6. Roofline of the BabelStream Dot Kernel (Dot outlines represent different implementations) (CUDA - Green, GCC - Light Blue, Purple - CRAY, Orange - NVHPC, Clang - Red)

An interesting observation we saw for all the BabelStream kernels was that NVHPC and CLANG had a higher Arithmetic Intensity compared to the CUDA and CRAY implementations.

Diving in the SASS we saw that both NVHPC and CLANG perform more IMAD (Integer Multiply and ADD) and ISETP (Integer and Compare Set Predicate) and BRA (Relative Branch) instructions

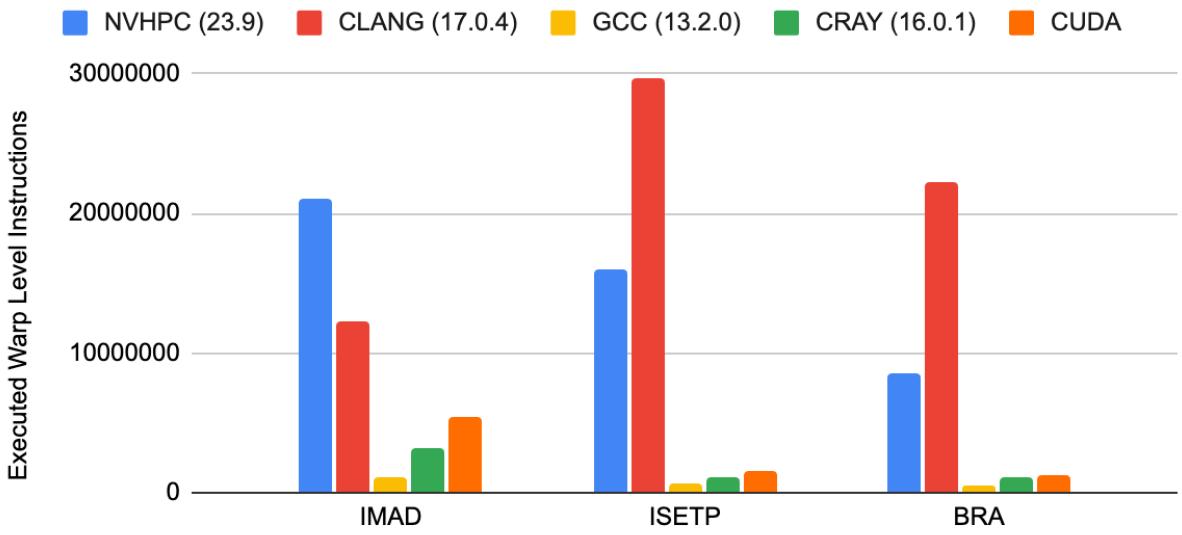

Figure 7. Instruction Statistics for IMAD, ISETP, and BRA

We believe that the performance disparity might be linked to the execution of additional instructions compared to CUDA and CRAY. However, drawing firm conclusions requires a more comprehensive investigation, including an examination of the generated intermediate representation (IR) or possibly conducting mini-benchmarks on these kernels.

5.4. Getting Closer to the CUDA Baseline

We observed that OpenMP implementations struggled with the dot kernel, a simple vector reduction task. In contrast, the CUDA implementation of the dot kernel utilized shared memory to store partial sums for each launched block, resulting in a better achieved bandwidth

The OpenMP implementations took a different route for reduction, relying on warp-level shuffle instructions instead of more shared memory like the CUDA version. We suspect that using shuffle instructions might not provide significant advantages compared to the shared memory approach in CUDA. This is because the shuffle instructions increase the number of registers allocated per thread, limiting the total threads that can be launched. Additionally, the throughput of shuffle instructions is lower compared to reading from shared memory and performing additions.

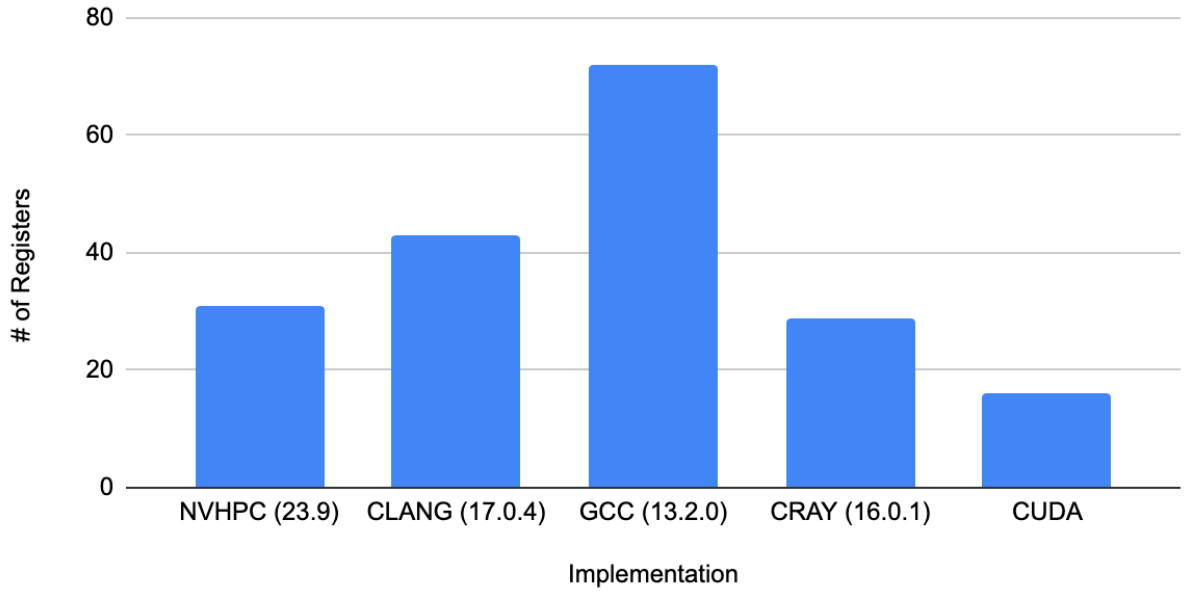

Figure 8. Registers Per Thread, BabelStream Dot Kernel

The CUDA version of miniBUDE's main kernel utilizes dynamic shared memory, a feature absent in all OpenMP implementations. Shared memory has much higher bandwidth and lower latency compared to global memory contributing to the faster runtime for miniBUDE.

6. Conclusion and Future Work

OpenMP is becoming more widely used to offload compute to GPUs due to its ease-of-use and portability compared to other models. The performance of OpenMP programs depends on the compiler. In this project, we evaluated the implementation of OpenMP 4.5 GPU-offloading on several compilers.

Our evaluation revealed significant performance disparities. CRAY emerged as a top performer, closely matching CUDA's baseline in BabelStream. However, GCC exhibited poor performance due to excessive memory allocations and high register usage. NVHPC and Clang showed middling performance, with potential areas for improvement. The observed differences underscore the importance of compiler choice as various implementations will prioritize different aspects of GPU acceleration. Developers must consider their specific requirements to find an optimal compiler.

In the future, we hope to address specific implementation issues. In addition, due to the time constraints on the project, we only evaluated these compilers on NVIDIA A100 GPUs on the Perlmutter supercomputer. Further work should be done to evaluate these compilers on other GPU architectures across a variety of machines.

7. References

[1] NVIDIA. [n.d.]. CUDA® Parallel Computing Platform. https://developer.nvidia.com/cuda-zone#:~:text=CUDA%C2%AE%20is%20a%20parallel,harnessing%20the%20power%20of%20GPUs.

[2] ROCm. [n.d.]. HIP: Heterogeneous-compute Interface for Portability. https://github.com/ROCm/HIP

[3] Khronos Group. [n.d.]. SYCL - Standard C++ for heterogeneous parallel programming. https://www.khronos.org/sycl/

[4] Dion Harris. 2021. NERSC Turns on World's Fastest AI Supercomputer. https://blogs.nvidia.com/blog/nersc-perlmutter-ai-supercomputer/.

[5] PassLab. [n.d.]. Parallel Programming and Performance Optimization With OpenMP. https://passlab.github.io/OpenMPProgrammingBook/openmp_c/1_IntroductionOfOpenMP.html

[6] OpenMP. [n.d.]. OpenMP Accelerator Support for GPUs. https://www.openmp.org/updates/openmp-accelerator-support-gpus/

[7] University of Bristol High Performance Computing. [n.d.]. BabelStream. https://github.com/UoB-HPC/BabelStream

[8] University of Bristol High Performance Computing. [n.d.]. miniBUDE. https://github.com/UoB-HPC/miniBUDE

[9] National Energy Research Scientific Computing Center. [n.d.]. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/

[10] GNU. [n.d.]. GNU Compiler Collection (GCC). https://gcc.gnu.org/

[11] LLVM. [n.d.]. Clang. https://clang.llvm.org/

[12] NVIDIA. [n.d.]. NVIDIA HPC SDK. https://developer.nvidia.com/hpc-sdk

[13] Hewlett Packard Enterprise. [n.d.]. Cray Programming Environments (CPE). https://cpe.ext.hpe.com/docs/cce/

[14] NVIDIA. [n.d.]. NVIDIANsightSystems. https://developer.nvidia.com/nsightsystems.