Evaluation of GPU-specific device directives and multi-dimensional data structures in OpenMP

by

Arijit Bhattacharjee

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ali Jannesari, Major Professor
Gurpur Prabhu
Myra Cohen

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

I would like to dedicate this thesis to my parents and grandparents without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Ali Jannesari for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Gurpur Prabhu and Dr. Myra Cohen. I would additionally like to thank Mohammad Norouzi from TU Darmstadt for his guidance throughout the initial stages of my graduate research without whom this research would have been without an inception.

ABSTRACT

OpenMP target offload has been in the inception phase for some time but has been gaining traction in the recent years with more compilers supporting the constructs and optimising it at the general level. Its ease of programming compared to other models makes it quite desirable in the industry. This work investigates how different compilers are interacting with the different constructs at runtime and how the callbacks affect the performance of each compiler. We also dive into the programs in the Polybench benchmark test suite with the main focus on linear algebra to generate an OpenMP GPU target offload implementation with parallelization techniques obtained from DiscoPoP for the C Language. The main focus is on the DOALL and reduction loops which come under loop level parallelism. The problem encountered are issues with the device data mapping. We also analyze the compiler used and see its behaviour in code generation for OpenMP target offload. While converting these benchmarks, we faced a myriad of issues related to the data mapping of multi-dimensional data structures onto the target from the host while using the GCC compiler. The main work done to counter this issue was to suggest a code transformation algorithmic approach which efficiently resolves the issues without loosing the correctness of the said programs.

CHAPTER 1. OVERVIEW

GPU acceleration is an effective way in high performance computing in the modern world. GPUs have been used in a wide variety of operations which require computational acceleration.The concept of using the multicore chips present for parallel computation was introduced in 1997 with the dawn of the OpenMP API. OpenMP allowed developers to parallelize their work directly using the all the CPU cores present. Normally a sequential algorithm runs on a single core of the processor. OpenMP allowed a significant increase in performance with only a slight tweak in the code. OpenMP introduced a feature to offload intense computations onto the GPU for faster compute times. An important step during the parallelization with OpenMP is selecting the right constructs (e.g., worksharing loop or task) and inserting them at the right position into the program. The goal is to achieve maximum speedup without violating correctness. The concept of offloading the intense parallel computation onto a GPU is studied as OpenMP has a feature which lets us do this efficiently. Offloading is a challenging concept both from the compiler and application developer standpoint. The significance of this study is to show how programs with multidimensional arrays can be parallelized with minimal effort in OpenMP GPU compared to other GPU based programming constructs. The jist of this study is that it provides a study on the aspects of performance and about how different parallel programming models behave when provided with the same use-case. It would also help future researchers understand the advantages and drawbacks of these models in order to improve them and also help them decide which one to use to get the maximum efficiency. We focus mostly on GCC due to its higher adoption rate among users and for the ease of setting up the environment required for the execution for target offload. We didn't explore the area of OpenCL due to the fact our hardware was mostly NVIDIA based cards and CUDA was well optimised for it. The reason we chose OpenMP as the model because it has become a mainstream parallel programming model which has a good support for

loop and task parallelism. It is a productive incremental programming model which is supported by most compilers and platforms.

CHAPTER 2. REVIEW OF LITERATURE

2.1 Hardware Architecture

CPU (Central Processing Unit) and GPU (Graphical Processing Unit) excel in different applications due to it being designed differently. Hardware design plays an integral part in their performance

2.1.1 CPU

CPUs were conceptualized for execution of serial code. A lot of technological advancements stretched the idea of Moore's law to a limit where it started slowing down. This led to a thinking into a whole new direction to make speedup possible. This led to the development of multi-core processors which opened a portal for parallel execution of code. The concept of memory hierarchy also changed the way the original design was envisioned.

In the present, we have a reached the bleeding edge of where technological prowess can take us. The number of cores in a multi-core processor is highly dependent on the task it is made for. The number of cores in personal computers are relatively lower compared to workstations. Personal computers may have somewhere near 16 cores but workstations have around 64 to 128 core systems. The cores themselves contain multiple units with different areas of responsibility. This includes units for the execution of program instructions, units for management and control, and memory units. (3)For improved access latencies, the memory is organized into a hierarchy consisting of registers, multiple cache- levels (e.g. L1, L2, and L3) and the system memory. The caches are different in size and performance, with smaller caches having lower latencies but also less capacity. Their number, their size, and how they are shared between the cores, depends on the particular CPU model.

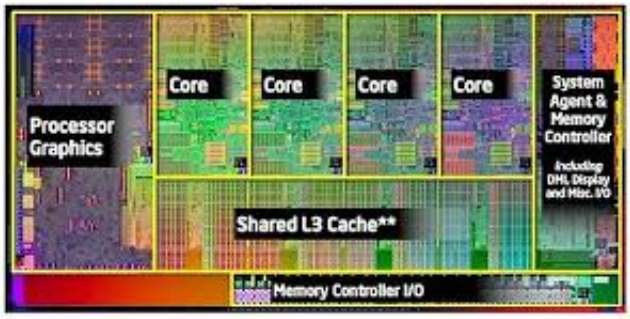

Figure 2.1 CPU architecture overview (4)

CPUs also have a very low compute density with a complex control logic. It also has some large caches inbuilt and are optimised for serial operations with fewer execution units and higher clock speeds. They also have a very shallow pipeline which is less than 30 stages and have a low latency tolerance. Newer CPUs have a higher degree of parallelism possible.

2.1.2 GPU

GPUs were primarily made for the acceleration of computer graphics. GPUs are not focused on the fast execution of serial code but rather on performing a lot of independent tasks in a parallel pattern. The potential benefits of GPGPU spurred a huge wave of hardware and software developments. The general principle of the GPU architecture is followed by all manufacturers. GPUs have a hierarchical based model. The multiple units inside the GPU are classified as GPCs (Graphic Processing Clusters), TPCs (Texture Processing Clusters) and SMs (Streaming Multiprocessors). The SMs themselves contain many different units, which comprise, for example, single-precision cores, double-precision units (DPUs), load/store units, and special function units (SFUs) used to perform fast approximate calculations of functions. Different GPU models have different characteristics. If a GPU has a lower number of DPUs compared to single precision cores can have huge complications in the throughput of the double precision instructions. There are a multitude of different units also inside the SMs. These include memory units such as

registers,cache and control units such as warp schedulers which manage the dispatch of instructions.

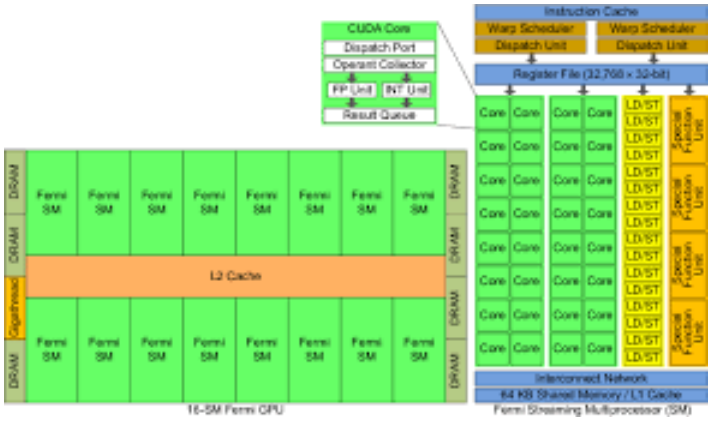

Figure 2.2 GPU architecture overview (5)

GPUs on the other hand have a higher compute density and can do a large number of computations per memory access. GPUs are primarily built for parallel operations with many parallel execution units with graphics being the best example. GPUs also consist of deep pipelines with hundreds of stages in them. They have a high throughput and also a high latency tolerance. Newer GPUs have a better flow control logic and also take advantage of scatter/gather memory accesses. There has been an elimination of one-way pipelines in the newer architectures.

2.2 OpenMP

OpenMP is a widely supported API used for sequential program parallelizations. No special code has to be written for different operating systems and platforms which is the main benefit of using OpenMP compared to other libraries designed for that purpose. All things being equal, the developer just requirements to clarify the parallelizable segments with the right OpenMP mandates a significant part of the rest, including information design and work appropriation, is taken care of by the upheld compilers. The OpenMP region is then executed in parallel by the forked threads, after which the sequential execution resumes on the master thread.

Figure 2.3 OpenMP Workflow (6)

2.3 OpenMP Target Offload

The accelerator support for OpenMP arrived the version 4.0 with significant revisions and extensions made in the version 4.5. OpenMP 5.0 promises another set of major revisions but it has not been made available to the public as to date with development still focusing on the promises of OpenMP 4.5. The directives involved are very similar to what OpenACC offers but they are not the same. The current usable implementations of OpenMP target offload are offered by Cray, IBM, LLVM/clang and GCC compilers. There has been support for both AMD and Nvidia based GPUs but the majority of the community focuses on the Nvidia GPUs as the compatibility of the compilers are better. The OpenMP device model is a host centric model with one host device with multiple target devices of the same type with the device being a logical execution engine with local storage. The device data environment is associated with a target data or target region. The main construct which is the target construct controls how data and code is being offloaded to a device.

2.3.1 OpenMP 4.0+ Execution Model

The target region is the basic offloading construct in OpenMP. It defines the section of the program which is offloaded to the device. When a target region is encountered, the code it contains is executed on a device and by default, the code inside the target region executes sequentially. At the end of the target region, the host thread waits for the target region code to finish and then continues executing the next statements.

```
#pragma omp target
    structured block
```

Listing 2.1 Target Pragma

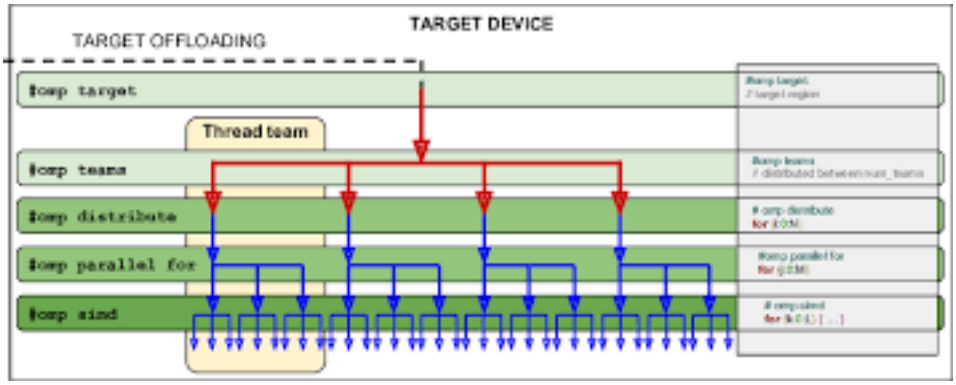

Figure 2.4 OpenMP Target Offload Execution Model (10)

The host and the device have separate memory spaces and in order to access the data inside the target region, it must be mapped to the device. The mapped data must not be accessed by the host until the target region has completed execution. The default behaviour in OpenMP 4.5 is that the scalars referenced in the target construct are treated as firstprivate which is a new copy on the device with the value initialised on the host. Also the static arrays are copied to the device on entry and back to the host on exit. The map clause is used to gain more control on the data mapping on the target construct.

```
#pragma omp target map(map–type:list)
```

There are different map-types available to the user for implementation. **to** is used to copy the data to the device on entry. **from** is to copy the data to the host on exit. **tofrom** is to copy the data to the device on entry and back on exit. **alloc** is used to allocate an uninitialised copy on the device but doesn't copy the values.

Keeping the data resident on the device is one of the important aspects for target offload as moving data between host and device is expensive on a lot of the current hardware present. The general idea would be to avoid mapping data in every target region if it can be kept on the device between target regions. **target data** constructs just maps the data and does not offload any of of the code with **target update** constructs copying values between host and device between target constructs.

The parallelism of the device is different as GPUs behave differently to CPUs. GPUs don't support a full threading model outside of a single streaming multiprocessor (SM). There is also no synchronization or memory fences possible between SMs. There is also no coherency present between the L1 caches. The parallel region inside a target region will only execute on one SM. Comparing it to CUDA, it can only synchronize threads inside a thread block, and not between thread blocks.

To counter these issues, the **teams** construct was introduced. This creates multiple master threads inside a target region with each master thread spawning its own team of threads within a parallel region. The threads in different teams cannot synchronize with each other. The concept of barriers, critical regions, locks, atomics only apply to the threads within a team. As ever, loops are the main source of parallelism in most applications and especially so for GPUs. If we offload a parallel loop to the device, we would like to distribute the iterations of the loop across the teams as well as across the threads within the teams. The **distribute** construct can be used to do this. It is like the **for** construct but it assigns the iterations of the following loop to different teams. The **device** clause can be used to explicitly specify which device to offload to. By default, the host thread blocks all the threads until the target region is completed but the **nowait** clause can be used to change the behaviour.

2.4 DiscoPoP

DiscoPoP (Discovery of Potential Parallelism)(2) is a software tool that is designed to aid in the detection and implementation of parallel patterns. It is a joint project between TU Darmstadt

and Iowa State University. It combines static and dynamic analysis to address the shortcomings of previous approaches, which were often too conservative in the identification of parallelizing opportunities.

The basic idea is that a sequential program is analyzed during multiple stages in which certain parallel patterns are first identified and then ranked based on the expected benefits from parallelization. Implementation suggestions for those patterns in the form of OpenMP constructs are provided to aid the user.

2.4.1 Workflow

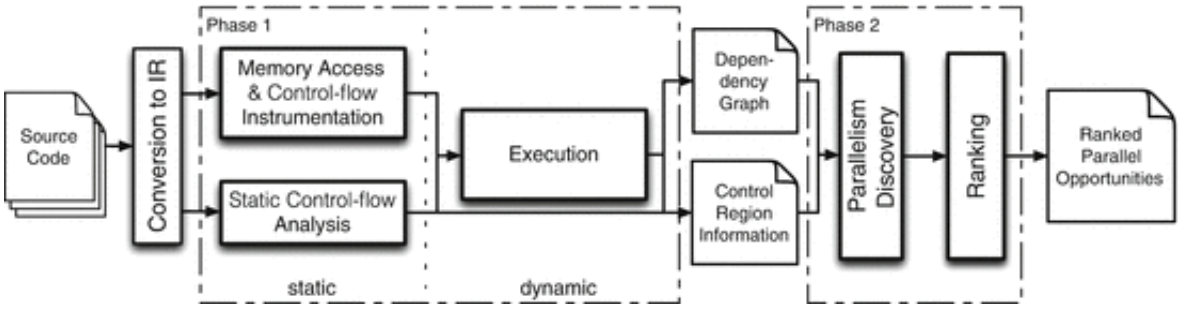

Figure 2.5 DiscoPoP Workflow (2)

The workflow of DiscoPoP can be divided into three stages. In stage 1, data regarding the said program is collected to be be analyzed with the second stage identifying possible parallel patterns with the last stage ranking them.(1)

2.4.1.1 Static and Dynamic Analysis

The primary stage can be additionally partitioned into a static and a powerful investigation stage. Subsequent to creating LLVM IR from the source code of the successive program, the static investigation stage starts. Here, a static control-stream examination is performed and a few pieces of the program, for example, memory gets to, are instrumented. The subsequent instrumented program is then executed in the wake of being connected with the DiscoPoP library, which executes the entirety of the instrumentation capacities. In this powerful stage, a dynamic-stream

control examination is performed and information is gathered, including data about dependency conditions.

2.4.1.2 Detection of Parallel Patterns

The results from the first stage are enough to detect the parallel patterns present. From the resulting data, we can construct a CU graph whose nodes are computational units(CUs), which are connected by edges representing the different data dependencies present. This graph is then mapped onto the Program Execution Tree(PET) of the program. Upon analyzing these structures, the possibility arises to detect different parallel patterns such as DOALL loops.

2.4.1.3 Ranking the Patterns

In this final stage, the patterns which were detected previously are then ranked taking in the attributes. The ranking order is then determined based on how promising the expected benefits from the parallelization.

2.4.2 Parallel Patterns

Even for reduction loops, benefits from execution on the GPU can be expected: although the maximum number of threads is lower than the loops number of iterations, a lot of parallelism is still achievable. DiscoPoP has the ability to detect a multitude of patterns such as tasks and pipelines but a closer look is only taken at the DOALL and reduction loops which come under loop level parallelism. These types of loops are the most suitable for achieving significant performance benefits on the GPU.

2.4.2.1 DOALL loops

DOALL parallelism is present where statements inside the loop dont have any inter-iteration dependence. This type of parallelism is preferred while executing in parallel as each thread can be

provided an iteration of the loop over a substantial number of cores which are like present in a GPU.

```
for(int i=0; i<N; i++){
 C[i]=A[i]+B[i]
}
```

Listing 2.2 DOALL Pattern

In the example we see that there is no inter-dependence between the iterations as different memory spaces are accessed in the different iterations.

2.4.2.2 Reduction Loops

Reduction loops are a little different than DOALL patterns as they allow certain dependencies to be present for the reduction variables. The reduction operator in such kinds of operations must be associative and commutative which enable the execution in any order.
This example shows us how reduction variables are combined with the array elements in the presence of a reduction operator.

```
sum=0;
for(int i=0; i<N; i++){
 sum=sum+A[i]
}
```

Listing 2.3 Reduction Pattern

Executing all the iterations in parallel is not possible as the same memory is both written and read. So to counter this issue, intermediate calculations are performed in parallel to combine elements pairwise, which is then replaced until all elements are combined. The correctness of the

result will hold true due to the properties of the associative and commutative reduction operations.

2.5 Polyhedral Model

The polyhedral model is a powerful framework for automatic optimization and parallelization.(8) It is based on an algebraic representation of programs, allowing to construct and search for complex sequences of optimizations. This model is now mature and reaches production compilers. The polyhedral model is a semantical, algebraic representation which combines analysis power, transformation expressiveness and flexibility to design sophisticated optimization heuristics. . It was born with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations. The polyhedral model is closer to the program execution than operational/syntactic representations because it operates on individual statement iterations, or statement instances. It has been the basis for major advances in automatic optimization and parallelization of programs

2.5.1 Static Control Parts

Static Control Parts (SCoP) are a subclass of general loops nests that can be represented in the polyhedral model. A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time). The iteration domain of these loops can always be specified thanks to a set of linear inequalities defining a polyhedron.

2.6 Unified Memory

The concept of unified memory was also fascinating and looks quite promising if it is ever included in the OpenMP API by the consortium. It can be achieved as of this date by explicitly changing the OpenMP runtime in the LLVM framework with a modified version of cudaMallocManaged, which is the CUDA equivalent for using unified memory being used. The results obtained in that study show the advantages of on demand data migration to the device. It

also shows how data reuse and the complexity of the data structure used can have a significant impact in the performance of the model. Our study focuses more on the current compiler optimizations that are present for use to the general public. Unified memory can help remove the compiler limitations that are present for usage of shared memory.

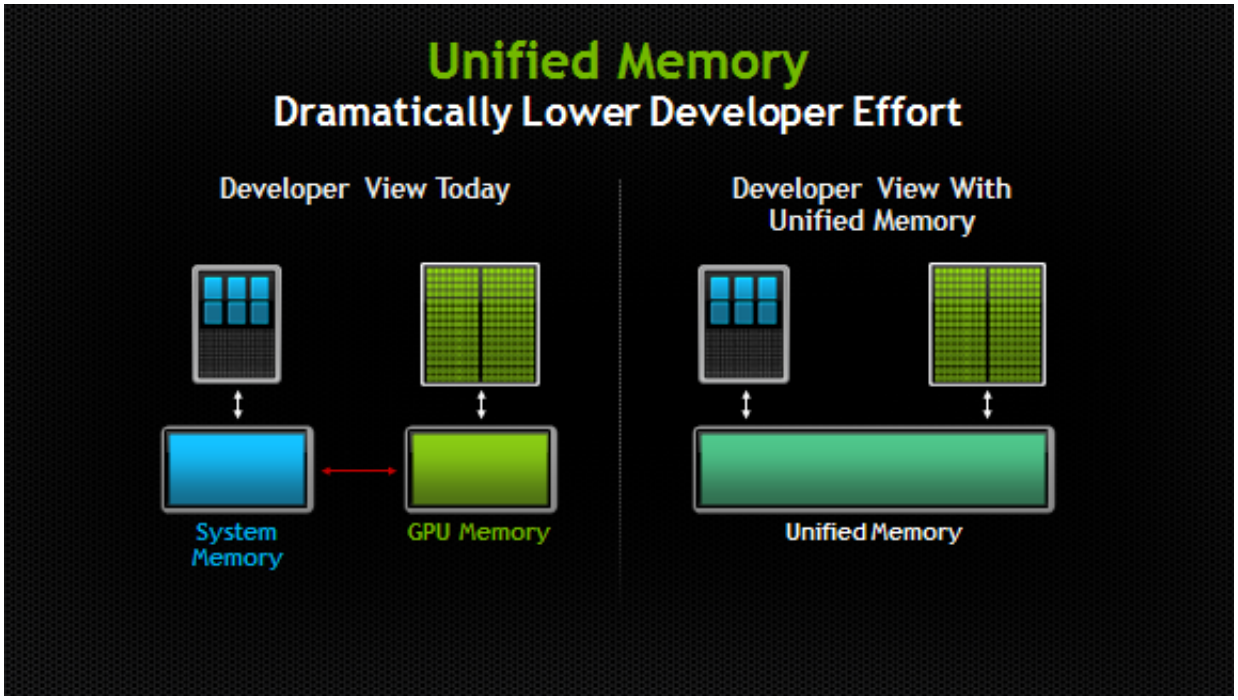

Figure 2.6 Unified Memory (7)

CHAPTER 3. PROBLEM DEFINITION & APPROACH

3.1 Problem Description

Upon directly converting the DiscoPoP provided parallelization techniques into the GPU based directives didn't work for the GCC compiler. We were facing 3 specific errors that were being thrown by the compiler.

```
libgomp: cuCtxSynchronize error: illegal memory access
libgomp: cuMemFreeHost error: illegal memory access
libgomp: device finalization failed
```

Listing 3.1 Memory Access Failures

The first error is a context management error. cuCtxSynchronize is a flag which blocks the device until all preceding tasks have been completed. cuMemFreeHost tries to free memory space pointed to by its call. The last error means that that the GPU device couldnt be configured properly in order to have the offload of data. All of the errors do say illegal memory access which goes to say that the data mapping of memory for the offload onto the device is not taking place in a proper manner. Mapping of memory is the most important part of the offload in order to get correct results. GCC 9.1.0 doesn't allow for the user to allocate directly on the shared memory. In short, mapping the data onto the device is not taking place efficiently and thus causing the issue for the GCC compiler.

3.2 OpenMP target offload runtime analysis

A study does depict some unique patterns noticed when evaluating the target offload directives, but for the sake of accuracy, we also performed a set of experiments similar to this study. The NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit was used to profile the various target offload directives and to analyze their behaviour during code generation. While

looking at the target directive running on the GCC compiler, we did notice that CUDA calls dominate during runtime. The allocation and de-allocation of memory was the most time consuming of all. While doing the same thing for the clang compiler, we did notice that CUDA memory allocation for the target directive was not existent. Most of the time is spent in the OpenMP runtime and a noticeable overhead was witnessed when the mapping of memory took place. Clang uses a set of CUDA pointer manipulation functions while directives like team distribute are being called. GCC on the other hand had a huge chunk of time dominated by CUDA based calls for the proper execution of these directives.

3.3 Code Transformation Algorithm

Upon rigorous brainstorming with different methods and ideas, we were able to device a code transformation algorithm for how we can implement multi-dimensional arrays for OpenMP target offload. We analyze the linear algebra programs in the kernels section in Polybench for our study.

3.4 Approach

In this study, we are looking at the DOALL loops which are present in the programs. A DOALL loop is a type of loop level parallelism where each iteration is independent of any previous iteration in the same loop. Also Polybench as a benchmark is made of static control parts(SCoP).Static Control Parts (SCoP) are a subclass of general loops nests that can be represented in the polyhedral model. A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time)(8). We are are concerned with the calculations which occur only in this region. The example shown to demonstrate the algorithm is of a two dimensional array and is similar for any n dimensional array

Algorithm 1 Sequential to Parallel

Input : A Sequential program with nD arrays
Output : An OpenMP GPU implementation of the program
FUNC(SEQ TO PAR)
{ Use DiscoPoP to evaluate the presence of DOALL and reduction loop level parallelism
//Initialize array
DATA_TYPE* arr
arr = $(DATA\_TYPE*)malloc(x * y * sizeof(DATA\_TYPE))$;For 2D
arr = $(DATA\_TYPE*)malloc(x * y * z * sizeof(DATA\_TYPE))$;For 3D
//Flatten array
$arr[i][j] -> arr[i * x + j]$For 2D
$arr[i][j][k] -> arr[x + i * (y + j * z)]$For 3D
//Mapping Memory
#pragma omp target enter data map(alloc:arr[])
//Kernel Run
#pragma omp target map(from:arrtmp[] to:arr[])
#pragma omp teams distribute parallel for schedule(static,1), collapse()
//Kernel Run END
#pragma omp target exit data map(from:arr[])
//END
}

3.4.1 Assisted Parallelization

DiscoPoP is what you call an assisted parallelization tool. It provides the user with possible parallelizable constructs for sequential programs which can be used as per their discretion. An example of a DOALL loop being identified and DiscoPoP providing the OpenMP equivalent pragma is given below.

```
{"id":"1:26","startline":35,"endline":37,"type":"DoAll"
,"pragma:":"#pragma omp parallel for","private":["i","j"]
,"shared":["A.addr"],"firstprivate":["ni.addr","nk.addr"]
,"reduction":[],"lastprivate":[]}
```

Listing 3.2 Sample Polybench Parallelization Suggestion

3.4.2 Arrays to pointers

The first step would be to look at the array definitions. Rather than considering them two dimensional arrays, we analyze them as pointers having a defined memory. This change results in the data being in a single stream making it easy to parallelize. The DATA TYPE over here is double and is defined in the Polybench header files.

```
DATA_TYPE* A;
A = (DATA_TYPE*)malloc(I*J*sizeof(DATA_TYPE));
```

Listing 3.3 Array Initialization

3.4.3 Memory Access

The next step is we look at how GPUs like to have the data defined. We know that in the end, CUDA libraries are being used to communicate with the GPU as the compiler binds itself with the ptxas assembler when we set up the environment. When the data is being passed onto the device, an operation similar to cudaMemCpy is taking place. Generally in CUDA, allocating and copying a doubly-subscripted C array doesn't work. cudaMemcpy expects flat, contiguously

allocated, single-pointer, single-subscript arrays. Taking cue from the behaviour in CUDA we modify the allocation definitions. The former shows us the original Polybench implementation and the latter shows how our method is implemented. Our implementation is reading the matrix in a row wise configuration to effectively maintain memory coalescence. This change is needed to be done where the initialized arrays are being populated with data for the first time. GPU computations prefer contiguous memory locations as access to elements become more direct and this can be done by this method. Also this also enables each slice of the array to live on a different cache line thus eliminating false sharing.

```
for (i = 0; i < ni; i++)
    for (j = 0; j < nk; j++)
      A[i][j] = ((DATA_TYPE) i*j) / ni;
  for (i = 0; i < nk; i++)
    for (j = 0; j < nj; j++)
      B[i][j] = ((DATA_TYPE) i*(j+1)) / nj;
  for (i = 0; i < nl; i++)
    for (j = 0; j < nj; j++)
      C[i][j] = ((DATA_TYPE) i*(j+3)) / nl;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nl; j++)
      D[i][j] = ((DATA_TYPE) i*(j+2)) / nk;
}
```

Listing 3.4 Original Polybench Implementation

```
for (i = 0; i < ni; i++)
    for (j = 0; j < nk; j++)
      A[i*ni+j] = ((DATA_TYPE) i*j) / ni;
  for (i = 0; i < nk; i++)
    for (j = 0; j < nj; j++)
      B[i*nk+j] = ((DATA_TYPE) i*(j+1)) / nj;
  for (i = 0; i < nl; i++)
    for (j = 0; j < nj; j++)
      C[i*nl+j] = ((DATA_TYPE) i*(j+3)) / nl;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nl; j++)
      D[i*ni+j] = ((DATA_TYPE) i*(j+2)) / nk;
}
```

Listing 3.5 Transformed Code Implementation

3.4.4 Data Mapping Correctness

There has to be correct mapping of the memory taking place before the transfer of data is done.A deep knowledge is also needed to understand how the application behaviours for an efficient port of all the data. Memory mapping is critical for achieving optimal performance. Mapping complex data structures is quite error prone. It is a challenge in itself to have the proper mapping between the host and device. OpenMP has a stand alone directive which is the target enter data map construct. This construct lets us allocate memory onto the device data environment before the transfer or calculation takes place. We explicitly specify how much memory is needed to be allocated on the device for all the data structures. Also, there is no dynamic allocation of memory on the device while implementing the offload. All the memory definitions required by the GPU is sent before the kernel is executed.

```
#pragma omp target enter data map(alloc:)
```

Listing 3.6 Allocate memory on device

and approach another name While looking at the calculations being done inside the SCoP we again have to map with respect to how the data structures are interacting.

```
#pragma omp target device(0) map(from:) map(to:)
```

Listing 3.7 Map data onto device, and transfer back to host after execution

We also try to keep the data resident on the device for the majority of the time and only remove it after the kernel has finished execution. We can explicitly have control using the target exit data map construct.

```
#pragma omp target exit data map(from:)
```

Listing 3.8 Deallocate memory space

3.4.5 Computation offload

OpenMP for CPU uses a flat threading model whereas for GPUs, it uses a hierarchical thread organization. There exists a two-level parallelism as there are multiple thread blocks and each thread block consists of multiple threads. The concept of using the teams construct is beneficial for this model. One team is equivalent to one thread block. We use the distribute clause with teams to populate all the threads within the teams. We also use the collapse clause due to the presence of nested loops. Collapse clauses have shown to improve performance and is the same case here.

```
#pragma omp teams distribute parallel for private()
 schedule(static,1) collapse(2) firstprivate()
```

Listing 3.9 Sample Target Offload

3.4.5.1 Memory Access Coalescence

We explicitly specify a static scheduling with a chunk size of 1 so that consecutive memory locations are accessed by the consecutive threads. Code optimization is achieved by doing this increasing the chunk size results in a degradation in performance.

CHAPTER 4. PERFORMANCE EVALUATION

4.1 Environment and Hardware Setup

For doing any of the parallel based computation, setting up the environment correctly is considered a milestone in itself. For our implementation, we use GCC 9.1.0 which is mated with OpenMP 4.5 and parts of the OpenMP 5.0 feature set. Linking the GPU with GCC was done with a custom script which sets up the GCC source tree and builds the nvptx side of GCC. The script also builds the host GCC to bind with the GPU. Also the script builds the necessary assembler and linking tools for the implementation. For the LLVM compiler, we use clang 10rc2 and it is also bound to the LLVM NVPTX backend(12) for OpenMP target offload. The CUDA version used is CUDA 10.1. Setting up the clang for OpenMP target offload was a tedious process. We are using the Polybench 3.2 test suite.(13)

For the GPU, we use a Nvidia Geforce Titan X Pascal(14) with 12 GB of DDR5X memory and a total of 3584 CUDA cores with CUDA Compute 6.1. For the CPU, we use a 8 cores 16 threads AMD Ryzen 7 2700X @ 3.7 GHz with a max boost of 4.3 GHz.

4.2 Testing Results

For our evaluation we use two sizes of data sets. The size of the data set is defined by the specifications inside each head header file of the respective benchmark. The Standard problem size was 1024 which is depicted by the blue bar graph and the Extra large was 4096 which is depicted by the red bar graph for atax,2mm,3mm,gemm, syr2k and mvt. The common factor is that for the CUDA implementation, a thread block of (32,8) was provided after multiple manual adjustments in order to maximize load balancing. Also for the OpenMP implementation, we used the DiscoPoP provided parallelization techniques distributed over all the available threads in our system which is 16 in our case. Flag -O3 was used for static code optimization in order to

improve efficiency. The alpha and beta values were kept constant for all the scenarios. The results that are obtained are taken on the average of 10 runs on each of the platform. We are focused on the timings of these computations.

4.2.1 atax

atax is Matrix Transpose and Vector Multiplication. We see that the OpenMP target offload on GCC takes extra time. This is due to the dominant presence of single dimension arrays in the kernel. This program is less compute intensive and fewer number of instructions.

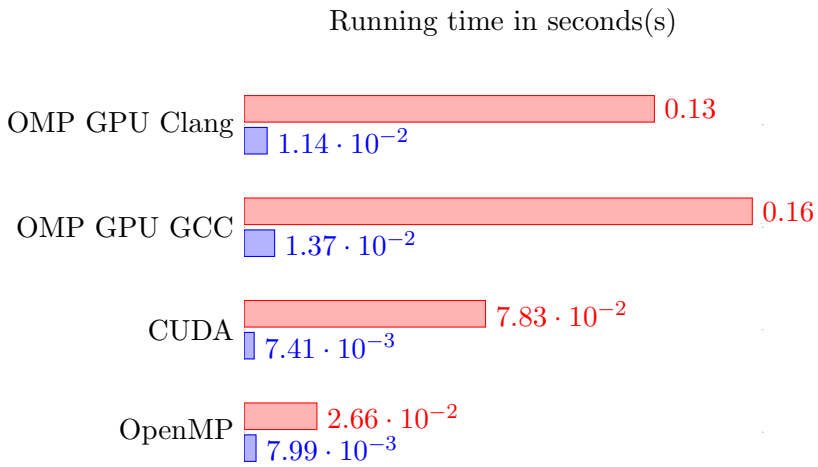

Figure 4.1 Test results for atax

4.2.2 2mm

2mm is 2 Matrix Multiplications (D=A.B; E=C.D).The implementation in CUDA is the fastest among all. Our implementation is competitive to the results in CUDA and shows a clear improvement over the multi-core CPU implementation. The benchmark is compute intensive due to the presence of multiple 2D arrays. The performance is also dependent on the quality of code generation done by the compiler.

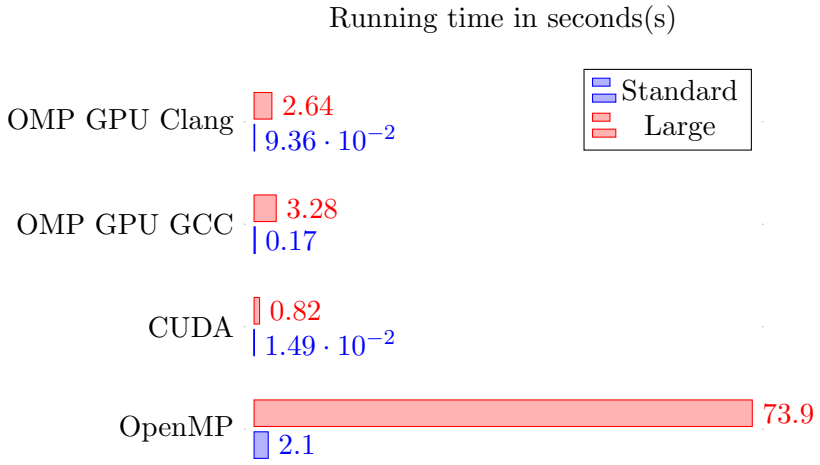

Figure 4.2 Test results for 2mm

4.2.3 3mm

3mm is 3 Matrix Multiplications (E=A.B; F=C.D; G=E.F).It is a similar result here. CUDA proves to be the most efficient overall. The clang version performs in a similar way that it outperforms the GCC counterpart. Our implementation comes in a close second showing its usefulness and a significant performance boost over the OpenMP version. It has similar properties to 2mm.

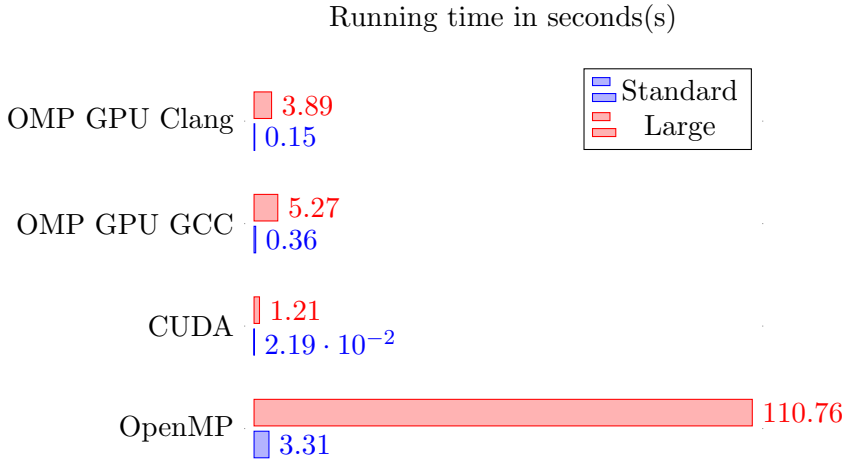

Figure 4.3 Test Results for 3mm

4.2.4 gemm

gemm is Matrix-multiply C=alpha.A.B+beta.C.In this benchmark we observe the similar trend that CUDA is faster than all the other platforms whatsoever be the problem size. OpenMP GPU has a good performance boost compared to OpenMP CPU.

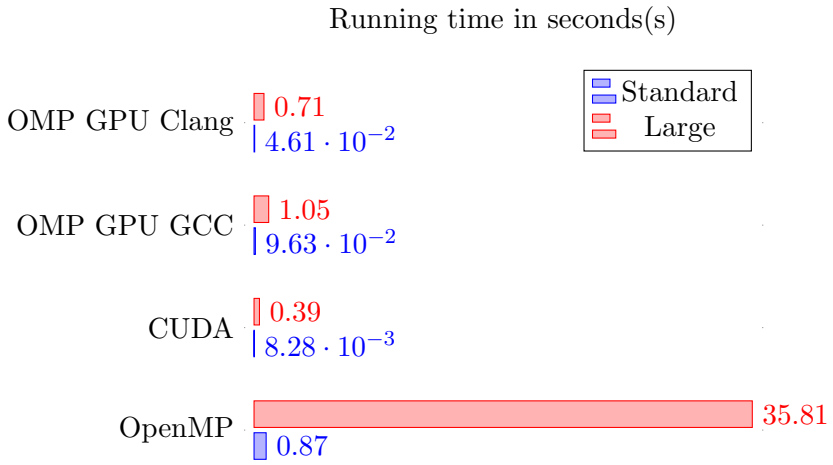

Figure 4.4 Test Results for gemm

4.2.5 syr2k

sry2k is Symmetric rank-2k operations.The results obtained from this were also very similar to the previous ones. The pragma directive in the calculation which is inside the SCoP has a clause for collapse(3). This is used as there are 3 loops in a nested configuration which are being used for the calculation. This loop collapse into a single nested loop causes a reduction in loop overhead and is providing the improved run-time performance.

4.2.6 mvt

mvt is Matrix Vector Product and Transpose. This program is very similar to atax and the behaviour shown also reflects that. This program is not that compute intensive hence CPU model is faster. We also analyzed this benchmark as well as atax through Nvidia Nsight in order to

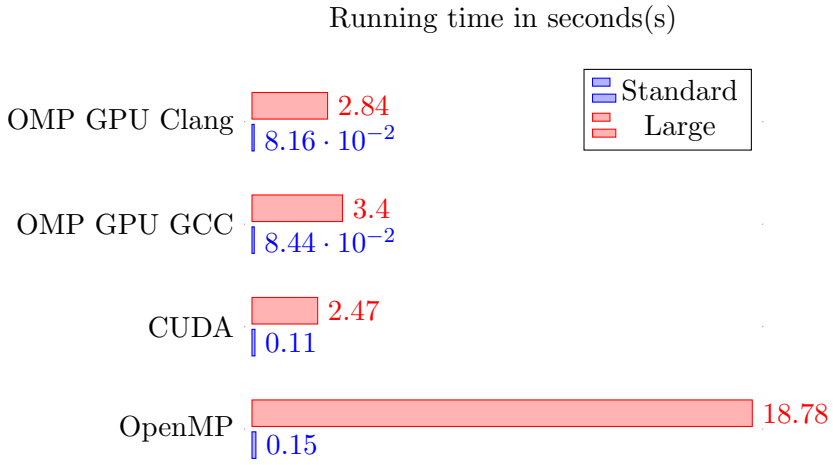

Figure 4.5 Test results for syr2k

know why it behaved like this. Most of the time taken over here was due to the data mapping that takes place while copying onto the device.

4.3 Code Characteristics

This section shows us how each of the models we tested compare to themselves from the programmers point of view in the design phase as well as the profiling phase. The table provides us with how many lines of code have been modified from the sequential model. For finding the effort quantitatively (17) we use the below equation to find the programming effort necessary to parallelize the program. We need the values of LOC-total which is the total number of lines of code and LOC-frac is the number of lines written in the respective programming model for the particular program. The regular C statements are only considered in the total count.

$$Effort[\%] = 100 * \frac{LOC_{frac}}{LOC_{total}} \tag{4.1}$$

On average we did see the effort coefficient to be around 33% for the CUDA based programs. Comparing that to OpenMP which had a coefficient of 9.4% does show the extra lines of codes needed for converting the sequential program to CUDA. OpenMP GPU for target offload using our implementation averaged around 15%.The coefficient OpenMP GPU for target offload on

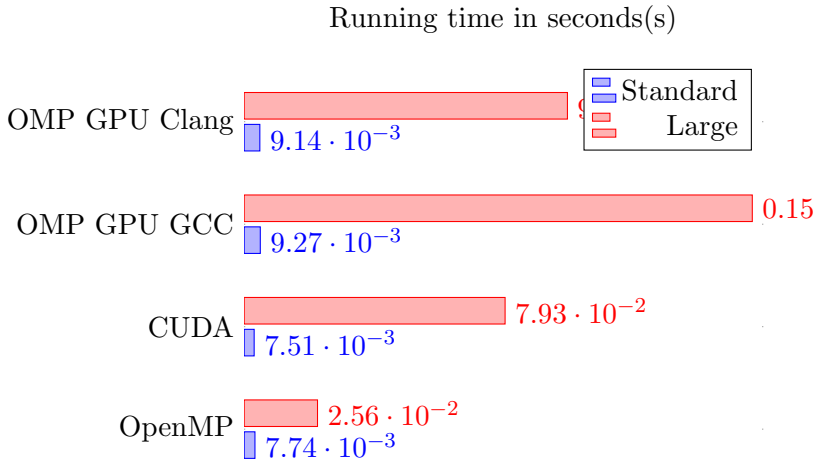

Figure 4.6 Test Results for mvt

clang averaged lower than the GCC counterpart as only the new directives had to be added as clang behaves differently.This does show us that OpenMP GPU for target offload could be a viable solution for larger computations in the future for its ease of programming by users.

CHAPTER 5. CONCLUSION

This study concludes that we could successfully port the Polybench benchmark test suite onto the GPU using OpenMP target offload directives for the GCC compiler. It depicts that CUDA based computations are still quicker to complete as it is well optimized by Nvidia. This work also showed that parallel codes are much more efficient when there is more data to go through. This research also provides a code repository for Polybench for OpenMP GPU that will be open-source to all researchers who may need it to analyse its performance counters. Till the time OpenMP 5.0 is released, this is the method we will have to use and considering the fact that there is work still being done on OpenMP 4.5 to optimize it, it will still take a long time for OpenMP 5.0 to come out and be efficient. We were also mainly focused on rectangular loops for this study.

5.1 Future Work

Our work was mostly focused on the kernels in the linear algebra section of Polybench but it could be also applied to the other benchmarks present. This algorithm could also be used in some large benchmarks like NAS (18) or even LULESH which are some real world physics simulation programs. We also did some work with Rodinia and over time we could have another base to build upon. We also feel that this algorithmic approach can also be used for non-polyhedral model loops too. SYCL could be also something which would be interesting to look at. We could even have a look into the Cray compiler and PowerPC based CPUs to analyze this offloading behaviour. A major work on AMD GPUs could also be undertaken. OpenMP ARB should come out with an official support for Unified memory and deep copy for pointer translation.

REFERENCES

[1] Mohammad Norouzi, Felix Wolf, Ali Jannesari: Automatic Construct Selection and Variable Classification in OpenMP. InProc. of the International Conference on Supercomputing (ICS), Phoenix, AZ, USA, pages 330341, ACM, June 2019.

[2] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, Felix Wolf: DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities. InTools for High Performance Computing 2014, Proc. of the 8th Parallel Tools Workshop, Stuttgart, Germany, October 2014, chapter 3, pages 3754, Springer International Publishing, 2015

[3] John L Hennessy and David A Patterson. Computer Architecture: A Quantitative Approach; 6th ed. Cambridge, MA: Morgan Kaufmann, 2019.

[4] https://www.theregister.com/Print/2012/06/28/review_amd_and_intel_mainstream_cpus/

[5] Huang, Miaoqing Men, Liang Lai, Chenggang. (2013). Accelerating Mean Shift Segmentation Algorithm on Hybrid CPU/GPU Platforms. 10.1007/978-1-4614-8745-6_12.

[6] https://computing.llnl.gov/tutorials/openMP/

[7] https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[8] Benabderrahmane MW., Pouchet LN., Cohen A., Bastoul C. (2010) The Polyhedral Model Is More Widely Applicable Than You Think. In: Gupta R. (eds) Compiler Construction. CC 2010. Lecture Notes in Computer Science, vol 6011. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-11970-5_16

[9] https://developer.nvidia.com/CUPTI-CTK10_1u1

[10] Monsalve Diaz, J.M., Friedline, K., Pophale, S., Hernandez, O., Bernholdt,D., Chandrasekaran, S.: Analysis of openmp 4.5 ooading in implementa-tions: Correctness and overhead. Parallel Computing 89, 102546 (08 2019) DOI: https://doi.org/10.1016/j.parco.2019.102546

[11] Alok Mishra, Lingda Li, Martin Kong, Hal Finkel, and Barbara Chapman. 2017. Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading. In LLVM-HPC17: LLVM-HPC17:

[12] Gheorghe-Teodor Bercea, Carlo Bertolli, Arpith C. Jacob, Alexandre Eichenberger, Alexey Bataev, Georgios Rokos, Hyojin Sung, Tong Chen, and Kevin OBrien. 2017. Implementing implicit OpenMP data sharing on GPUs. In Proceedings of Fourth Workshop on the LLVM Compiler Infrastructure in HPC, Denver, CO, USA, November 1217, 2017 (LLVM-HPC17:), 12 pages. DOI: hps://doi.org/10.1145/3148173.3148189

[13] http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

[14] https://www.techpowerup.com/gpu-specs/titan-x-pascal.c2863

[15] Solihin, Yan (2016). Fundamentals of Parallel Architecture. Boca Raton, FL: CRC Press.

[16] zen, Gray, S. Atzeni, M. Wolfe, Annemarie Southwell and Gary Klimowicz. OpenMP GPU Offload in Flang and LLVM. 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) (2018): 1-9.

[17] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Koodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC '17). ACM, New York, NY, USA, 1-6. DOI: https://doi.org/10.1145/3110355.3110356

[18] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The Nas Parallel Benchmarks. Int. J. High Perform. Comput. Appl. 5, 3 (September 1991), 6373. DOI:https://doi.org/10.1177/109434209100500306

[19] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, C. H. Still, Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In Proceedings of 27th IEEE International Parallel Distributed Processing Symposium, Boston, MA, pages 1-14

[20] Kim, Jae-Jin Lee, Seok-Young Moon, Soo-Mook Kim, Suhyun. (2010). Comparison of LLVM and GCC on the ARM platform. 10.1109/EMC.2010.5575631.