An Empirical Study of Parallelizing Test Execution Using CUDA Unified Memory and OpenMP GPU Offloading

Taghreed Bagies  
King Abdulaziz University, Jeddah, Saudi Arabia  
Iowa State University, Ames, Iowa, USA  
tbagies@iastate.edu

Ali Jannesari  
Iowa State University, Ames, Iowa, USA  
jannesari@iastate.edu

Abstract—The execution of software testing is costly and time-consuming. To accelerate the test execution, researchers have applied several methods to run the testing in parallel. One method of parallelizing the test execution is by using a GPU to distribute test inputs among several threads running in parallel. In this paper, we investigate three programming models CUDA Unified Memory, CUDA Non-Unified Memory, and OpenMP GPU offloading to parallelize the test execution and discuss the challenges using these programming models. We use eleven benchmarks and parallelize their test suites by using these models. Our study shows some limitations (e.g. cache size, branch divergence, and load imbalance) when using GPUs to execute the testing in parallel.

Keywords—Software Testing, Parallel Test Execution, GPU Offloading, Unified Memory, OpenMP, CUDA.

I. INTRODUCTION

In software testing, a test suite has a number of unit cases that test each component of a program. The number of unit tests could be quite large (e.g. Google Test Automation Platform has 150 million test runs in an average day [1]). Running unit tests in parallel can significantly speed up the test execution time [2], [3]. Therefore, many researchers use distributed environments such as cloud computing or virtual machines (VMs) to accelerate the test execution [4], [5], [6], [2], [7], [8], [9]. Since these proposed methods are costly in terms of maintenance, energy consumed, and time [10], two studies suggest utilizing GPUs to distribute test inputs in several threads running in parallel to accelerate the test execution [10], [11].

However, these two studies, which apply GPUs for parallel test execution, evaluate the performance in very specific embedded software that is not general-purpose (e.g. do not handle complex data type or loops). They implement a test suite by using two programming models (CUDA and OpenCL) to run test inputs in GPUs. These two programming models require hard coding for data transfer between CPU and GPU memories, which might affect the performance of the test execution in GPUs.

GPUs Unified Memory (UM) addresses the challenges of data transfer by enabling automatic data migration between GPU and CPU memories [12]. It supports on-demand page migration which has two major advantages: 1) The process of copying complex data structures is simplified, and 2) UM makes it possible to run kernels with memory tracks larger than the GPU memory capacity [13].

OpenMP 4.5 is a directive-based programming model that is useful to create reusable software without the need to learn about the programming paradigm or the hardware [14]. It supports heterogeneous systems with an offloading feature such that it parallelizes an application in GPUs. It has a broader user community, is more widely available, and has a growing number of developers [13].

Nevertheless, there is a lack of knowledge of how some features (e.g. loops, different data types, and arrays) of a program will affect its test execution in GPUs. For example, branch divergence yields when different threads execute different paths, which serializes the threads execution [10]. Different test inputs having a different number of loop iterations may increase the branch divergence. Different arrays sizes of test inputs may negatively impact memory access. Also, some challenges may impact the performance of test execution when using OpenMP GPU offloading. For example, a deep copy is a challenge in OpenMP GPU offloading [13] [14]. If test inputs are dynamically allocated in memories, OpenMP GPU offloading might be error-prone and complicate the implementation of a test suite. Without using UM, developers need to handle data transfer explicitly, which may limit the amount of data that can be transferred and increase the time needed to transfer these data. With UM, data migrations will happen during kernel run time which may increase the overall execution time of testing. However, no studies have yet determined which model is better with respect to parallelizing test execution in GPUs. Besides, some of these programming models may have different API calls (e.g. creating thread context, launching kernel) in the backend, which could affect the overall execution time of the testing.

In this paper, we conduct a performance study to investigate the challenges discussed previously when parallelizing test execution in GPUs. We use eleven programs from different domains (see Table I). These programs have different loop structures (e.g. different levels of nested loops) and have a different number of arrays and data types. For each
selected program, we implement five versions of test suites:
1) Sequential, 2) OpenMP multicore (OMP-CPU), 3) CUDA Non-Unified Memory (CUDA-NUM), 4) CUDA Unified Memory (CUDA-UM), and 5) OpenMP GPU Offloading (OMP-GPU). For each test suite version, we evaluate the performance in terms of execution time, data transfer, branch divergence, and memory dependency.

The following are the main contributions of this work:
- Provides an empirical evaluation of CUDA-UM, CUDA-NUM, and OMP-GPU to parallelize the test execution in GPUs.
- Analyzes and compares the results of numerous programs with a multicore CPU.

II. RELATED WORK

The parallelization of software test execution has been applied by two groups of researchers. The first group implements a test driver that reads test inputs from a file and uses a GPU to run the testing in parallel [11], [10]. In [11], they conduct a performance study of test execution in GPUs by using CUDA-NUM. Yaneva et al. [10] propose a compiler-assisted framework that automatically generates an OpenCL code from a C sequential code and executes the tests in parallel on a GPU. These two studies did not use UM and OMP-GPU which are recently introduced.

For evaluation, both studies use Embedded Microprocessor Benchmark Consortium (EEMBC) [15]. Here, we investigate CUDA-NUM, CUDA-UM, OMP-GPU to implement a test driver and we use three different domains of benchmarks (Table I).

The second group of researchers use cloud computing or VMs to execute a test suite in parallel [9], [4], [6], [7], [2], [5] [8]. However, a test suite could have a test dependence and could affect the parallelization of a test suite [16], [17]. In contrast, we use a file containing test inputs. Each input represents a possible value used to test a program and has different path coverage from the other inputs. When parallelizing the test execution, each test input will be executed by a thread with no shared data between different threads. Therefore, there is no dependency on the order of different test inputs.

III. EXPERIMENTAL SETUP

The goal of the experiment is to explore and evaluate three programming models (CUDA-NUM, CUDA-UM, OMP-GPU) which can be used to test a program in a GPU. We evaluate the performance in terms of the execution time in GPUs versus multicore CPUs. To analyze the results, we use two NVIDIA profilers Nsight Compute CLI [18] and Nvprof [19] in the GPU.

For the CPU machine, we use x86_64 Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz. We implement the test suite in C. We use clock_gettime() [20] to measure the execution time in seconds. Our machine has a maximum of 36 cores such that each core has one thread. So, we run our experiment for OMP-CPU by using 36 threads and 18 threads. For the GPU machine, we use NVIDIA Volta architecture Tesla V100. We use cudaEventCreate() [21] to calculate the execution time in seconds. For Compiler, we use Clang9 to compile the sequential, OMP-CPU, and OMP-GPU test suite version. We use nvcc [22] for CUDA-UM and CUDA-NUM test suite version.

A. Benchmark

We could not choose any arbitrary benchmark since CUDA does not support some C standard libraries [23]. We must test a program without any modification in its source code. For example, we could not execute the test of a program that deals with string or reads from a file and writes to a file. We take this limitation into account and create a set of criteria to choose benchmarks from different domains. First, the source code of a selected program does not rely on any C standard libraries such as a string. Second, developers of a selected program provide test inputs or a clear specification (e.g. Readme file) for how to generate test inputs automatically. Lastly, selected programs should be different in their functions, arguments data type, number of arguments, and source code structure such as loops and
if-statements. Table 1 summarizes the benchmarks used in the study.

Polybench is a benchmark suite of numerical computations with static control flow in various application domains such as linear algebra computations and dynamic programming. Its source code is available in [24] and used for multiple code analysis research and benchmarkings such as parallelization and code transformation with OpenMP [25], [26]. The benchmark involves nested loops and handles arrays with a different number of dimensions and computations. Besides, the developers provide a clear specification about the minimum and maximum array size. They also provide a function that initializes arrays. Therefore, we choose two programs from different domains of Polybench. As shown in Table I, the selected programs are different in the number of arguments, and dimensions of arrays (e.g. Adi handles only two dimensions of arrays; Atax has a mix of one and two dimensions of arrays; reg_detect has a mix of two and three dimensions of arrays).

Since Github (a popular repository) is a practical way to obtain source code [27], we choose two programs from image processing field and clustering algorithm. We choose Image Manipulation Application [28] since it has many functions, many if-statements, and a combination of "while" and "for" loops. To generate the test inputs, we download different sizes of images and use them to test this program. The developers provide functions to convert images to arrays. For the clustering algorithm, we choose K-means [29] since the developer provides the set of test inputs, and it has a different source code structure than Polybench and Image Manipulation Application.

B. Test Case Inputs

Since the maximum number of threads per block in our GPU is 1024, we use 1024 test inputs. The test inputs are provided by developers for each selected benchmark. In GPUs, several threads are running in parallel as a group known as a warp. The warp size in our GPU is 32 threads/warp [30]. We distribute 1024 test inputs in several blocks based on the warp size. Therefore, the number of threads per block is 32 where the number of blocks is the number of test inputs (1024) divided by the number of threads per warp (32). We store test inputs in a text file such that each line represents a test input.

Since we are using OMP-GPU and CUDA-NUM, there is a limitation in the number of test inputs that could be transferred to a kernel. To transfer the test inputs from host to device, programmers need to use cudaMemAlloc() and cudaMemcpyp() for CUDA-NUM. One of the arguments (passed to these two functions) is input size (size_t that could be 32 bits or 64 bits depending on the machine). To transfer a float array, for example, the input size is calculated as the following: 

\[
\text{size of an array} = \text{Number of test inputs} \times \text{sizeof(float)} \times \text{number of elements in the array}. \]

C. Profiling

To evaluate the performance of all test suite versions, we run each version several times and compute the average of their execution times. Since all selected programs have loops and arrays, the branch divergence and memory access may affect the performance of the test suite versions in a GPU. To analyze the performance between CPU and GPU, we use Nsight Compute CLI [18] to measure the effect of warp execution efficiency (for branch divergence) and stall memory dependency. Also, we use nvprof [19] to analyze the effect of data migration and deep copy with respect to the three programming models in a GPU. Nvprof provides the absolute execution time of transferring data, launching kernel, and invoking different APIs.

IV. TEST SUITE IMPLEMENTATION

For each selected program, we implement five versions of a test suite (Sequential, OMP-CPU, CUDA-UM, CUDA-NUM, OMP-GPU). A test suite has three components: 1) data Structure to store test inputs, 2) read function to get data from test inputs file, and 3) launch function to execute the test.

V. RESULTS

We validate the outputs of each test suite. All the versions generate the same outputs, except four programs (Adi, Atax, Ludcmp, Syr2k) of Polybench. These four programs give slightly different outputs because the precision of the floating points in GPUs is different from CPUs [31]. The Nsight Compute CLI profiler measures two metrics: 1) warp execution efficiency, and 2) stall memory dependency. The results of these two metrics are almost the same from different programming models because they are related to the hardware architecture of GPUs. For seek of simplicity, we report one result from CUDA-UM (that is similar to CUDA-NUM and OMP-GPU).

A. CPU Multicore vs. GPU Performance

Figure 1 shows the speedup of the five test suites over the sequential version. OMP-CPU with 36 threads has significant speedup among the other versions. There are three
main reasons that cause test suites in a GPU to be slower than in a multicore CPU.

1) **Cache size:** The cache size in our CPU machine is 20 MB whereas our GPU machine has 128KB cache size. The cache size of the GPU negatively affects the performance of all test suites since all programs under test have arrays with different dimensions that access memory frequently for a read and write operation.

The effect of data access pattern for the temporal and spatial locality is worse on the GPU than on the multicore CPU. For the test suite in the GPU, each thread will handle an instance of a test input that may include different arrays. Each thread accesses different arrays’ positions within the same instruction. For example, Adi has memory access for three different arrays in a single statement. This increases the number of misses. As a result, it has the lowest speedup ratio.

2) **Load Imbalance:** In GPUs, several threads execute in parallel as a group known as a warp. Each thread will execute a test input that has a different array size from the other test inputs. Some test inputs, for example, might have array sizes between 50-500 (executed in parallel by one warp) while other test inputs could have an array size of 500-2000 (executed in parallel by another warp). This leads to a warp load imbalance such that some warps execute large array sizes while others execute small array sizes. As a result, some warps will be inactive waiting for the other warps to finish their job.

One of the stall reasons that makes warp to be inactive is the stall memory dependency that is measured by Nsight Compute CLI. As shown in Figure 2, almost all Polybench programs have stall memory dependency 50% and higher. These programs are loop-oriented and deal with different dimensions of arrays.
Table II

RESULTS OF ABSOLUTE EXECUTION TIMES OF TRANSFERRING DATA BETWEEN HOST (CPU) AND DEVICE (GPU) FOR THE THREE TEST SUITE VERSIONS (S). DtoH (DEVICE TO HOST); HtoD (HOST TO DEVICE).

<table>
<thead>
<tr>
<th>Program</th>
<th>CUDA memcpy DtoH</th>
<th>CUDA memcpy HtoD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CUDA-UCM</td>
<td>CUDA-UM</td>
</tr>
<tr>
<td>Adi</td>
<td>4.40E-02</td>
<td>5.05E-02</td>
</tr>
<tr>
<td>Atax</td>
<td>1.92E-03</td>
<td>2.48E-03</td>
</tr>
<tr>
<td>Correlation</td>
<td>2.64E-01</td>
<td>4.19E-01</td>
</tr>
<tr>
<td>Durbin</td>
<td>1.79E-03</td>
<td>3.34E-03</td>
</tr>
<tr>
<td>Dynprog</td>
<td>3.01E-06</td>
<td>8.10E-06</td>
</tr>
<tr>
<td>Jacobi-2d-imper</td>
<td>1.78E-01</td>
<td>1.23E-01</td>
</tr>
<tr>
<td>ludcmp</td>
<td>1.78E-03</td>
<td>2.34E-03</td>
</tr>
<tr>
<td>reg_detect</td>
<td>1.46E-03</td>
<td>1.76E-03</td>
</tr>
<tr>
<td>Syr2K</td>
<td>1.14E-01</td>
<td>1.20E-01</td>
</tr>
<tr>
<td>Image Manipulation</td>
<td>4.53E-02</td>
<td>7.57E-02</td>
</tr>
<tr>
<td></td>
<td>9.34E-03</td>
<td>7.86E-02</td>
</tr>
</tbody>
</table>

Figure 2. Stall Memory Dependency expressed as percentage.

Table III

RESULTS OF ABSOLUTE EXECUTION TIMES IN SECONDS OF KERNEL FOR THE THREE TEST SUITE VERSIONS (S).

<table>
<thead>
<tr>
<th>Program</th>
<th>CUDA-UCM</th>
<th>CUDA-UM</th>
<th>OMP-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adi</td>
<td>2.70E+02</td>
<td>2.71E+02</td>
<td>3.06E+02</td>
</tr>
<tr>
<td>Atax</td>
<td>2.79E+00</td>
<td>3.34E+00</td>
<td>4.05E+00</td>
</tr>
<tr>
<td>Correlation</td>
<td>3.25E+02</td>
<td>3.29E+02</td>
<td>3.97E+02</td>
</tr>
<tr>
<td>Durbin</td>
<td>7.66E-01</td>
<td>1.07E+00</td>
<td>8.35E-01</td>
</tr>
<tr>
<td>Dynprog</td>
<td>1.19E+02</td>
<td>1.19E+02</td>
<td>1.76E+02</td>
</tr>
<tr>
<td>Jacobi-2d-imper</td>
<td>3.75E+02</td>
<td>3.76E+02</td>
<td>4.31E+02</td>
</tr>
<tr>
<td>ludcmp</td>
<td>4.24E+02</td>
<td>4.25E+02</td>
<td>4.02E+02</td>
</tr>
<tr>
<td>reg_detect</td>
<td>5.32E+01</td>
<td>5.32E+01</td>
<td>6.53E+01</td>
</tr>
<tr>
<td>Syr2K</td>
<td>2.07E+02</td>
<td>2.07E+02</td>
<td>2.81E+02</td>
</tr>
<tr>
<td>Image Manipulation</td>
<td>8.67E-01</td>
<td>9.05E-01</td>
<td>1.14E+00</td>
</tr>
<tr>
<td>K-means</td>
<td>192.126</td>
<td>191.992</td>
<td>3.98E+01</td>
</tr>
</tbody>
</table>

3) Branch Divergence: Branch divergence arises when a group of threads in a warp executes different branches due to a condition statement. Divergent instructions will be serialized, leading to a significant performance loss [10]. When we test a program with arrays and loops, each thread will have different sizes of an array (different loop iterations). This increases the number of inactive threads per warp. For example, some threads will iterate 10 times while other threads will iterate 1000 times. In our study, all selected programs have loops, which increases the branch divergence in GPUs.

As shown in Figure 3, most of the test suites of selected programs have warp execution efficiency less than 45%. The test suite of the Atax program achieves more than 60% of warp execution efficiency because it has one nested loop iterating based on a 2D array. Unlike Atax, the other selected
programs in Polybench have more than one nested loop with different nested levels (two and three levels). Since the Image Manipulation Application program has the largest number of if-statements, it has the lowest warp execution efficiency.

B. Programming Models in GPU Performance

1) Effect of Transferring Data: In a GPU, using CUDA-UM provides a better speedup ratio than CUDA-NUM for three programs (Atax, Durbin, Image Manipulation). As shown in Table II, the total execution time of transferring data (from host to device and device to host) in these test suites is smaller in CUDA-UM than in CUDA-NUM, which provides a reason why these test suites achieve better speedup ratio in CUDA-UM than in CUDA-NUM.

Although OMP-GPU is easier to implement the test suite than CUDA-UM and CUDA-NUM, it has the lowest speedup ratio among all versions. As shown in Table II, transferring data for all programs is slower in OMP-GPU than CUDA-NUM and CUDA-UM. In OMP-GPU, all test inputs of a specific program will have the same array size since they are allocated statically due to the challenge of the deep copy. With CUDA-NUM and CUDA-UM, the array size of each test input will be allocated dynamically based on the input provided.

2) Kernel Effect: Table III shows that the execution times of OMP-GPU test suites of all programs (except Ludcmp) are bigger than CUDA-NUM and CUDA-UM test suites. This gives another reason why all test suites (except Ludcmp) of OMP-GPU has an overall slower execution time. In Ludcmp, the execution time of the kernel for OMP-GPU is slightly faster than CUDA-NUM and CUDA-UM whereas the execution time of transferring data in OMP-GPU is slower than CUDA-NUM and CUDA-UM (Table II). Therefore, the overall speedup ratio of these test suite versions is almost the same (less than 2.4).

3) Setting up the GPU Environment: Nvprof shows that other API calls also affect the overall execution time such as cudaMalloc, cudaFree, and cudaLaunchKernel. These API calls add extra time to the total execution times of the CUDA-NUM and OMP-GPU version. By using GM, GPU page faults affect the execution time of the kernel. Based on nvprof, the times of GPU page faults for all programs are close to their kernel's execution time. As shown in Table III, the differences in the execution time of the kernel between CUDA-UM and CUDA-NUM is not big. Furthermore, other API calls are invoked only in the OMP-GPU version such as cuCtxCreate, cuCtxDestroy, cuMemAllocHost, and cuModuleUnload. Some of these calls have more than one millisecond execution time. Therefore, these events affect the overall execution time in the OMP-GPU version.

C. Limitations

Table IV summarizes the limitations of the three programming models to execute testing in GPUs. For CUDA-UM, accessing data in UM increases the execution time of the kernel. Since we used Volta GPU that supports GPU page fault, On-demand migration, and Address Translation Service, the execution time of the kernel is not significantly bigger than CUDA-NUM for the majority of the selected programs. With regard to CUDA-NUM, we need to use two pointers (aliasing pointer, device pointer) to dynamically allocate an array that has two or higher dimensions. In CUDA-NUM and OMP-GPU, we were not able to transfer data at once test inputs that have size greater than 2GB. We need to distribute the inputs to different kernels or devices, which adds more effort to the implementation of their test suites. For OMP-GPU, we were not able to dynamically allocate test inputs with different array sizes. Thus, we statically allocate the data such that each array of a test input has fixed length. This increases the amount and execution time
of transferring data. Additional overhead API calls lead to an increase in the overall execution time of testing.

VI. DISCUSSIONS

It should be mentioned that GPUs are slow in I/O operations, i.e., for programs relying on reading and writing files, GPUs are not a good choice to execute the testing in parallel. Also, CUDA does not support some C standard libraries such as string operations as well as recursive function.

We use the functions that initialize the values of each test input provided by the developers of the selected programs. Thus, the used test inputs are considered representative tests and relevant to the selected programs.

In our experiment, we use 1024 test inputs (given the maximum number of threads per block is 1024 in a GPU) to explore the feasibility of the three programming models to parallelize the test execution in GPUs. The number of test inputs could be increased for further investigation. This is beyond the scope of our study because at this stage we are not focusing on the scalability (with respect to the number of test inputs) of different programming models in GPUs.

Since we are using OMP-GPU, we were able to use Ungrouped Arrays instead of Array of Struct (AoS). AoS will be complicated in terms of implementation if arrays have more than two dimensions. In addition, it will limit the number of transferring data. All test inputs will have the same size array when using Ungrouped Arrays even though some of them might actually have a smaller size than others.

Mishra et al. [13] were able to use unified memory with OMP-GPU because they flatten the arrays (e.g int* C = omp_target_alloc(N*M*sizeof(int))). In our experiment, we can not flatten the arrays of a test case input. If test inputs, for example, include a 2D array, we declare a 3D array in the test suite such that the row represents a test input whereas the column represents a pointer of the 2D array. Therefore, we cannot flatten this array as it would be complicated and error-prone to access an element of a flattened array.

We use Clang 9 to compile the OMP-GPU version. When we tried to use ”target enter data” to handle the deep copy similar to [14], the compiler failed to interpret and execute it. According to Diaz et al. [14], some compilers do not support some of OMP-GPU features in some systems. Although some offload features are supported by Clang 9 for OpenMP 5.0, there are other features are under implementations.

Although there is no dependency on the order of different test inputs, there is a validity threat if a program under test has public variables. Our experiment uses a program that does not include any public variables to mitigate shared public variables between different threads.

VII. CONCLUSION

This paper aimed to investigate three different programming models (CUDA-UM, CUDA-NUM, and OMP-GPU) to parallelize the test execution in GPUs. Our result shows that cache size in GPUs negatively impacts the performance due to the presence of arrays. Also, branch divergence and stall memory dependency influence the test suites’ performances. Besides, data migration has major effects in OMP-GPU than in CUDA-UM and CUDA-NUM because of the challenge of the deep copy. As a result, speeding up test execution via the GPU is a dead end. There is a qualitative reason (it only works for very specific systems under test) and a quantitative one (better speed up can be achieved via multi-core).

ACKNOWLEDGMENT

We would like to thank King Abdulaziz University, Saudi Arabia for supporting the Ph.D. scholarship of this research.

REFERENCES


