Accelerated Test Execution using GPUs

Ajitha Rajan[1], Subodh Sharma[2], Peter Schrammel[2], and Daniel Kroening[2]

[1]School of Informatics, University of Edinburgh, UK

[2]Department of Computer Science, University of Oxford, UK

[1]*arajan@inf.ed.ac.uk*
[2]*{subodh.sharma,peter.schrammel,kroening}@cs.ox.ac.uk*

ABSTRACT

As product life-cycles become shorter and the scale and complexity of systems increase, accelerating the execution of large test suites gains importance. Existing research has primarily focussed on techniques that reduce the size of the test suite. By contrast, we propose a technique that accelerates test execution, allowing test suites to run in a fraction of the original time, by parallel execution with a Graphics Processing Unit (GPU).

Program testing, which is in essence execution of the same program with multiple sets of test data, naturally exhibits the kind of data parallelism that can be exploited with GPUs. Our approach simultaneously executes the *program with one test case per GPU thread*. GPUs have severe limitations, and we discuss these in the context of our approach and define the scope of our applications. We observe speed-ups up to a factor of 27 compared to single-core execution on conventional CPUs with embedded systems benchmark programs.

1. INTRODUCTION

The number of tests needed to effectively validate any non-trivial software is extremely large. For instance, Yoo et al. [24] state that for an IBM middleware product used in their study, it takes a total of seven weeks to execute all the test cases, making overnight builds impossible. Much of the research in software testing over the last few decades has focussed on test suite reduction techniques and criteria (such as coverage) that help in identifying the effective tests to retain. This trend is particularly seen in regression testing and black-box testing, where numerous optimisation techniques—test case selection, test suite reduction, and test case prioritisation—have been proposed to reduce testing time [23, 19]. Even after applying these optimisations, test suites remain large and their execution is typically very time consuming. This puts an enormous strain on software development schedules.

We present an approach with the potential of executing test suites in a fraction of the original time and explore its feasibility on embedded systems benchmark programs. Our approach leverages the speedup offered by **Graphics Processing Units** (GPUs). GPUs are *massively parallel processors* featuring multi-threaded performance unmatched by high-end CPUs. The single-chip peak performance of state-of-the-art GPU architectures exceeds 1500 GFLOPS, which compares to around 100 GFLOPS for a traditional processor at its best [12]. A further advantage of GPUs is that they are more *energy efficient* than their multi-core CPU counterparts [11, 20]. Finally, in terms of cost per performance, GPUs are more *affordable* than multiple PCs. In particular, the management cost of one computer with GPU is much smaller than that of a corresponding cluster of PCs.

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09.
http://dx.doi.org/10.1145/2642937.2642957.

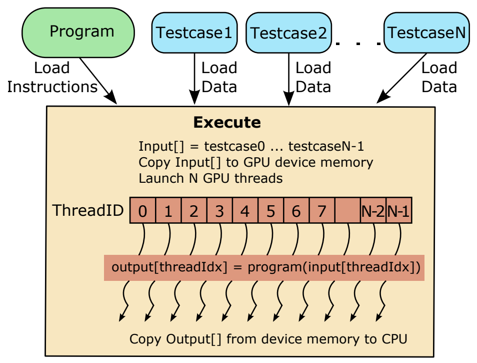
Program
Testcase1
Testcase2
TestcaseN
Load Instructions
Load Data
Load Data
Load Data
Execute
Input[] = testcase0 ... testcaseN-1
Copy Input[] to GPU device memory
Launch N GPU threads
ThreadID 0 1 2 3 4 5 6 7 N-2 N-1
output[threadIdx] = program(input[threadIdx])
Copy Output[] from device memory to CPU

Figure 1: Our Approach for Test Execution on a GPU

Test Execution using GPUs.

General-purpose computing on GPUs (GPGPU) has been successfully applied in a broad range of domains [15, 21, 8]. GPUs use a Single Instruction Multiple Thread (SIMT) architecture to exploit data-level parallelism. We believe software testing will benefit greatly from GPUs, and the most compelling reason for this is the fact that program testing, i.e., running the same program with multiple sets of test data, *naturally exhibits data parallelism that can be exploited with GPUs*. There has been no work in the past exploring this possibility and this paper paves the way in leveraging the acceleration provided by GPUs for software testing.

Existing literature on GPGPU investigates techniques to transform CPU versions of a program to run on the GPU. The key problem here is the identification of opportunities to parallelise. Our approach is dramatically different: we leave the program and its logic untouched and only paralellise *the running of multiple test cases* on the program.

Our approach simultaneously executes the *the program with one test case per GPU thread*. We will instrument the program with GPU device management code that acts as a wrapper for the original code. We store the test suite in the GPU device memory and launch one GPU thread with the original program functionality for each test input. The threads require no synchronisation or coordination since the executions of different test cases are completely independent. ***No program transformation*** from CPU program to GPU program is required in our proposed technique, and thus, we still test the original program logic and obtain results that are the same as those obtained with a CPU. However, current GPUs have severe limitations and impose restrictions on the class of programs we can test. We discuss these limitations in Section 3.3.

Our approach for accelerated test case execution using GPUs provides the following potential benefits: (1) a significant reduction in test suite execution time and as a result huge savings in testing costs, and (2) for the same allotted test time, allows more test cases to be executed, potentially increasing the likelihood of fault detection [9], and (3) better energy efficiency than testing using multi-core CPUs or PC clusters [11, 20].

Related Work.

GPUs can be exploited for non-graphical tasks using General Purpose computing on Graphical Processing Units (GPGPU) [15]. GPGPU has been successfully applied in a broad range of applications. There is growing interest in the software engineering community to use the massive performance advantage offered by GPUs. Recenty, Yu et al. explored the use of GPUs for test case generation [25]. Bardsley et al. have developed a static verfication tool, GPUVerify, for analysing GPU kernels. Li at al. and Yoo et al. have adapted multi-objective evolutionary algorithms for test suite optimization to execute on GPUs. Nevertheless, we are not aware of any existing study that has explored the use of GPUs to accelerate *test suite execution* which is the goal of our work.

Our contributions.

To validate our hypothesis on test acceleration with GPUs, we make the following contributions in this paper:

1. We describe and implement our approach for executing test cases in parallel on a GPU using the CUDA programming model.
2. We evaluate our approach experimentally using example programs and test suites, and discuss the achieved speedup. We use programs from the embedded systems domain, in particular programs from the embassy benchmark suite [6].
3. We discuss the limitations of our approach, i.e., the testing activities it can be applied to, and the application domain.

2. BACKGROUND

The success of GPUs in the past few years has been due to the ease of programming using the CUDA [1] and OpenCL [3] parallel programming models, which abstract away details of the architecture. In these programming models, the developer uses a C-like programming language to implement algorithms. The parallelism in those algorithms has to be exposed explicitly. The GPU SIMT architecture can deliver extreme speedups if the different threads executed have no data dependencies. We now present a brief overview of the core concepts of CUDA.

The highest level of the CUDA thread programming model is a *parallel kernel*. A kernel is a function that is invoked by a program running on the host CPU but is executed on the GPU. Kernel functions are identified by means of the keyword *__global__*. Kernel functions can only access the memory on the GPU; any data required by it has to be copied to GPU memory before invoking the kernel. The kernel is executed as a *grid* of *blocks* of *threads*. In other words, threads are grouped into blocks and blocks are grouped into a grid. Grid and block dimensions can be specified when launching the kernel. Grids and blocks can be multidimensional and along each dimension there is a hardware-defined limit on the number of threads and blocks that can be created.

The memory system is organized in three levels of hierarchy. Closest to the core are the *registers*, which have the lowest latency and are private to each thread. Next is *shared memory*, which is again small and has low access latency. Shared memory is available to the threads within a block. All threads can access the large *global memory*, which is comparatively slow (400–800 cycles).

Finally, the execution model in CUDA requires threads in blocks to be executed in groups called *warps*. A warp is a group of 32 threads in a block that are launched and executed together. All threads in a warp execute the same instruction but on different data. This is often referred to as *lock-step* execution semantics. When a conditional instruction is encountered with control flow divergence among the threads within a warp, GPUs resort to *predication*, which can improve instruction scheduling and cache behavior of programs. Each thread block is mapped to one or more warps. As a result, we choose the thread block dimension as multiple of the warp size. Warps within a thread block can execute independently.

The example in Figure 3 gives the definition of a kernel function, which is invoked as follows:

```
compute<<<1,NUM_TESTS>>>(device_inputs)
```

Here, 1 specifies the grid dimension and `NUM_TESTS` specifies the block dimension. A pointer to a block of GPU memory is passed as an argument to the function. Note that such memory is allocated and initialised by the host code using the `cudaMalloc` and `cudaMemcpy` functions. A unique thread id within a thread block is given by the system variable `threadIdx.x` (along a single dimension) and the block id is denoted by `blockIdx.x`. Thus, a thread id that is unique in the grid can be obtained by calculating $blockIdx.x \cdot blockdim + threadIdx.x$.

3. OUR APPROACH

We believe that the execution of a large test suite is a natural match for this architecture, since it requires executing the same program multiple times with different inputs. Running a program with a test case on a CPU typically involves loading the program and the test case into memory and executing the instructions with the loaded data. This is repeated for every test case in the test suite. Each test case run is completely independent of other test case runs. Also, all the executions are over the same program, albeit possibly not the same instructions, depending on the control logic in the program. On the GPU, our approach will launch as many GPU threads as there are test cases, where each thread executes the same sequential program with a different test case. Since executions of different test cases have no data dependencies, there is no need for any thread synchronisation in our approach.

The key points of our approach (Figure 1) are:

1. We vectorise the test inputs so that their dimension is the number of test cases in the test suite.
2. We copy the vectorised inputs from the host memory to the GPU device memory.
3. We then launch the kernel with the program functionality on the requisite number of GPU threads (ideally as many as there are test cases). Each GPU thread will operate on the same program but with different test data, using the unique thread id to identify the test case inputs to execute over.
4. We copy the program output from the GPU back to the CPU.

```
#define ARRAY_SIZE 9

void quickSort(int[], int, int);
int partition(int[], int, int);

int main(void) {
  // sample input array as test case
  int a[] = { 7,12,1,−2,0,15,4,11,9 };
  quickSort(a, 0, ARRAY_SIZE−1);
  return 0;
}
```

Figure 2: Harness for testing Quicksort with one test input

```
#define ARRAY_SIZE 9
#define NUM_TESTS 256

__device__ void quickSort(int[], int, int);
__device__ int partition(int[], int, int);

// GPU function
__global__ void compute(int *tests) {
  // The thread ID identifies the test case
  int test_case = threadIdx.x*ARRAY_SIZE;
  quickSort(tests+test_case, 0, ARRAY_SIZE−1);
}

int main(void) {
  int host_inputs[NUM_TESTS][ARRAY_SIZE] = {....};
  int *device_inputs;

  cudaMalloc((void **)&device_inputs, sizeof(host_inputs));
  cudaMemcpy(device_inputs, host_inputs,
                    sizeof(host_inputs), cudaMemcpyHostToDevice);

  // Number of blocks is 1 and number of threads per block is 256.
  compute<<<1, NUM_TESTS>>>(device_inputs);
  return 0;
}
```

Figure 3: CUDA test harness for Quicksort with 256 tests

Developers using our approach only need to understand the program interface, i.e., inputs and outputs. The rest of the program is used as a black box.

3.1 An Example

To better understand our approach, consider Figure 2, which gives a harness for testing a quicksort program. We omit the code for the quicksort implementation, which is unchanged in our approach. The original quicksort program is available at [5]. One test input is provided as an integer array and the quicksort function is called, given the array and its lower and upper array index bounds.

Figure 3 gives a CUDA harness for testing the quicksort procedure. The CUDA kernel has four functions: *compute*, *main*, *quicksort* and *partition*. The *main* function now features a test suite with 256 tests, which are stored in the *host_inputs* variable. The test cases in *host_inputs* are copied to *device_inputs*, which resides on the GPU device memory, using *cudaMemcpy*. Finally, the function *compute* is called by the kernel, which launches one block of 256 threads in parallel.

The function *compute* runs on the GPU and is identified as a kernel using the keyword *__global__*. A local variable *arr_inputs* is assigned a test case from the suite of tests in *test_inputs* parameter using the unique thread id and the size of the array. The function *quicksort* is called using the array input as before. The functions *quicksort* and *partition* run on the GPU, which is indicated using the *__device__* keyword. The bodies of the functions remain unchanged and are not shown.

The main code changes required for this example using our approach stem from: (i) transferring the test suite from the host to the device, (ii) adding a kernel function (*compute*) to be called on all threads, (iii) reading the test case from the test suite using the thread ID in the kernel function. The rest of the code largely remains the same. In particular, note that that the *quicksort* and *partition* functions remain unchanged. As a result we still test the original functions with our approach.

Compilers for GPU programs are highly specialised and do not support all features used in C/C++ programs. The limitations of GPUs and their implications are discussed in the next section.

3.2 Limitations of GPUs and Implications

GPUs can offer considerable acceleration. However, in turn, there are severe limitations:

L1 GPU programs have to copy data back and forth from the host memory to perform I/O or when GPU memory is exhausted. Data transfer between the GPU and host memory is slow due to the high latency of the interface. Furthermore, the typical bandwidth of accesses to GPU memory is two to three orders of magnitude higher than the bandwidth of transfers to host memory over PCIe.

L2 The different programming model–CUDA or OpenCL–often requires heavy, non-trivial changes to existing source code to leverage GPU performance.

L3 Control-flow branching in source code (using control structures like if-then-else statements) penalises GPU performance. GPUs execute groups of threads in lock-step. All threads that belong to the same group execute the same instruction but use different data. Lock-step execution is violated if the branch instruction diverges. This can impact performance negatively [13].

L4 While the typical memory bandwidth of GPUs is about five times higher than that of CPUs [14], GPUs are restricted by the fact that their bandwidth is shared among thousands of threads [18]. This is not a problem in applications like graphics where threads share large data sets that can be retrieved from the shared memory in blocks. In applications that do not share data, data transfers from the device global memory to each of the several thousand threads will be a bottleneck.

L5 The compiler for CUDA source files (NVCC [2]) processes them according to C++ syntax rules. As a result, some valid C (but invalid C++) code fails to compile. Full C++ is supported for the host code. However, only a subset of C++ is supported for the device code, as described in CUDA programming guide [1].

We will now discuss these limitations in the context of our approach. Limitations L1 and L4 are believed to be less of an issue in next generation GPUs. A recent keynote from the CEO of NVIDIA predicts the next generation of NVIDIA GPUs (Pascal, to be released in 2016) to bring larger memory size and bandwidth (using stacked memory), faster data transfer between CPU and GPU (5 to 12 times more) using NVLink, and smaller, more energy-efficient chips [10]. In our approach, we would ideally want to do a data transfer once from the CPU to the GPU and once the other way. However, system calls and other program features that CUDA/OpenCL cannot handle require data to be transferred more frequently. Handling system calls effectively on the GPU is an active area of research [22].

Limitation L2 is not an issue for our approach, since we are not transforming the program to run on the GPU. Instead, we only need to write a test wrapper in CUDA or OpenCL that launches for each test case a copy of the program on a thread. Little or no knowledge of the program logic is needed and the transformation is typically straightforward. We plan to automate the insertion of this test wrapper in the future.

Limitation L3 restricts the programs we can test with our approach. We hypothesise that for programs with heavy branching our approach will not produce a significant speedup. We have applied our approach to programs with different degrees of control flow divergence to test this hypothesis. It is expected that the impact of this limitation will reduce in future-generation GPUs, which will feature more sophisticated branch prediction logic. Limitation L5 restricts the program features CUDA/OpenCL can support on the GPU device. Unsupported features can be re-implemented for CUDA, but this requires program transformations, which may affect the correctness of the test execution. CUDA and OpenCL are evolving and future releases will support more features of C/C++. However, this limitation is currently the primary constraint for the scope of our approach.

3.3 Scope of our Approach

Our approach is best suited to:
(1) C++ programs that can be compiled for the GPU. Limitation L5 drives the set of programs that satisfy this constraint.
(2) C/C++ programs with limited system calls.
(3) C/C++ programs with limited control flow branching.
(4) C/C++ programs that do not exceed GPU memory size.
Future generation GPUs and CUDA/OpenCL compilers will potentially allow a wider application scope. In this paper, we use C programs from the embedded systems domain that satisfy the constraints mentioned above.

4. EVALUATION

We check the feasibility of our approach on C programs from the embedded systems domain. We evaluate the hypothesis that *test execution on GPUs is faster than on CPUs* on the example C programs and test suites. We also show that our approach does not alter the program functionality and that the test case outputs on the GPUs and CPUs remain the same. Finally, we discuss the overhead of data transfer between host and device. In our experiments, the CPU we use is an Intel Xeon processor with 8 cores at 3.07 GHz and 16 GB RAM. The GPU we use features the GTX 670 Kepler architecture, 960 cores at 1.07 GHz, and 2 GB device memory.

4.1 Benchmarks

We use four benchmarks in our evaluation:

1. Image decompression using inverse discrete cosine transform (idctrn01)
2. Fast Fourier Transform processing in the automotive area (aifftr01)
3. Inverse Fast Fourier Transform processing (aiifft01)
4. Brake-By-Wire System (bbw, 2473 LOC)

The first three programs are from the Embedded Microprocessor Benchmark Consortium (EEMBC), which provides a diverse suite of processor benchmarks organised into categories that span numerous real-world applications, namely automotive, digital media, networking, office automation and signal processing, among others [16]. The three benchmark programs that we have chosen are from the automotive category. The benchmark programs have test inputs associated with them. All the inputs are stored in a large data structure. The program only reads a small fraction of the test inputs in the data structure for one execution iteration. However, the program is executed iteratively several times, each time reading test inputs from a different location in the data structure. The output values from all executions are captured for both CPU and GPU executions and later compared to determine correctness of test execution using our approach.

Speedup				
#Tests	**idctrn01**	**aifftr01**	**aiifft01**	**bbw**
1024	10	8	18	2
2048	18	12	23	2
4096	26	15	25	2
8192	25	14	24	2
16384	27	15	25	2

Table 2: Speedup (CPU time/GPU time rounded) for the 4 programs with GPU block dimension of 64

The fourth benchmark is a brake-by-wire (BBW) system provided by Volvo Technology AB designed in Simulink [7]. C code was generated from it using Simulink Coder. The system consists of five components, four wheel brake controllers for sensing and actuating, and a main controller responsible for computing the braking torque. We generated random test vectors over the input ranges of the five inputs, rotations per minute for each wheel and state of the brake pedal. The values of the four brake torque outputs were captured for CPU and GPU executions and compared.

Similar to the example illustrated in Section 3.1, we added GPU device management code to run the program with one test case on each GPU thread. We did not make any changes to the code that implements the program functionality. The modifications on all programs were straightforward and easy to implement.

4.2 Experimental Results

We collect the following data: 1. Execution time on the CPU, 2. Execution time on the GPU for different grid and block dimensions, 3. Test Outputs on the CPU and GPU, 4. Device from/to host data transfer time for the GPU executions.

Table 1 gives the results obtained from test execution on the CPU and GPU. Column *Benchmark* contains the name of the benchmark used. Column *#Tests* is the number of tests run on the program. *Block Dim* and *Grid Dim* are the number of threads in a block and number of blocks in a grid, respectively. *GPU time* and *CPU time* are times taken on the GPU and CPU, respectively, to execute all the tests on the program. *Device-Host time* column represents the time spent on data transfers between CPU (host) and GPU (device). Finally, the *Outputs Match* column indicates whether test outputs from the CPU run and the GPU run match.

Table 2 gives the speedup achieved by executing tests on the GPU compared to the CPU for the different benchmarks. Speedup is computed by dividing the *CPU time* column by the *GPU time* column in Table 1.

4.3 Discussion

Speedup Achieved.

As seen in Table 2, the speedup achieved with our approach is 2 to 27 times, depending on the benchmark and test suite size. Speedups achieved for the EEMBC benchmarks, *idctrn01*, *aifftr01* and *aiifft01*, are high (10 to 27 times). A possible explanation for this is that control flow in all of these benchmarks is induced by `for` statements rather than `if-else` statements. Recall that in Section 3.3 we mentioned that control-flow divergence reduces GPU performance since groups of threads execute in lock step. Lock-step execution is impossible if branches diverge. In our examples, the `for` statements cause only very limited divergence in control flow and hence a high speedup is observed.

On the other hand, the speedup achieved for the *bbw* example is only two times, regardless of test suite size. The *bbw* code contains

Benchmark	# Tests	Block dim.	Grid dim.	GPU time (ms)	CPU time (ms)	Device-Host time	Outputs Match?
idctrn01	1024	64	16	1.77	17.70	0.21	Yes
idctrn01	2048	64	32	1.95	35.33	0.35	Yes
idctrn01	4096	64	64	2.71	70.51	0.67	Yes
idctrn01	8192	64	128	5.60	141.18	1.19	Yes
idctrn01	16384	64	256	10.56	282.24	1.9	Yes
aifftr01	1024	64	16	25.42	192.11	12.33	Yes
aifftr01	2048	64	32	31.02	383.56	22.98	Yes
aifftr01	4096	64	64	50.57	766.98	45.81	Yes
aifftr01	8192	64	128	108.98	1533.66	94.86	Yes
aifftr01	16384	64	256	208.94	3067.93	178.03	Yes
aiifft01	1024	64	16	10.75	190.76	12.36	Yes
aiifft01	2048	64	32	16.52	380.60	23.01	Yes
aiifft01	4096	64	64	30.01	760.89	45.53	Yes
aiifft01	8192	64	128	62.88	1521.73	90.78	Yes
aiifft01	16384	64	256	124.19	3044.81	190.26	Yes
bbw	1024	64	16	3.45	5.50	4.18	Yes
bbw	2048	64	32	4.29	10.46	8.12	Yes
bbw	4096	64	64	9.06	20.35	8.47	Yes
bbw	8192	64	128	19.98	40.41	15.83	Yes
bbw	16384	64	256	39.43	80.31	30.88	Yes

Table 1: Results on both the CPU and GPU from running the 4 benchmarks with different test suite sizes

heavy control flow branching with if-else statements. Diverging control flow and lock-step semantics cause instructions on different branches to wait and synchronise, which leads to higher GPU execution times and lower speedups. The CUDA version of the *bbw* benchmark contains a large set of thread local variables. NVIDIA's *Kepler architecture* (evaluations were performed on the card having this architecture) allows 255 32-bit registers per thread to be allocated for thread local variables. Excess variables are spilled over to the global memory. We confirmed that when *bbw* benchmark was evaluated on 32 threads, a spill of 264 bytes was observed. Accessing global memory is known to be at least an order of magnitude slower.

Notice that for all four programs, the speedup achieved remains the same beyond 2048 tests (for grid dimensions 16, 32 and 64). The likely reason for this is that the number of blocks and hence warps for a very large number of tests exceeds the maximum number of warps that can be scheduled on a streaming multiprocessor. It might also be that for larger number of tests (>2048 in our examples), there are not enough resources available to run all the test cases in parallel. As a result, some of the threads will have to wait and be scheduled later.

For all the benchmarks, we saved and compared the test outputs from the CPU and GPU for different numbers of tests. We found that in all the cases listed in Table 1, the test outputs from our approach matched the test outputs from CPU. Although this is not a proof of correctness of our approach, it does serve as an initial evidence of feasibility of test execution with GPUs.

Effect of Block and Grid Dimensions.

The user-supplied dimensions for grid and block play a crucial role in the runtime of a kernel. A larger block size will result in frequent context switching of warps in a block. While thread (and warp) context switching in GPUs is relatively lightweight, the effects become tangible when kernel execution is long. Current GPU cards allow a maximum of 1024 threads in a single-dimensional block (a hardware restriction). The log-plot in Figure 4 quantifies the effect of the block size on the benchmarks. For this plot, we fixed the number of test runs to 1024. Notice that as the block size approaches the limit of the hardware, the execution times worsen. All the benchmarks show an optimal execution time when the block size ranges between 16 and 64. A straightforward reason for this

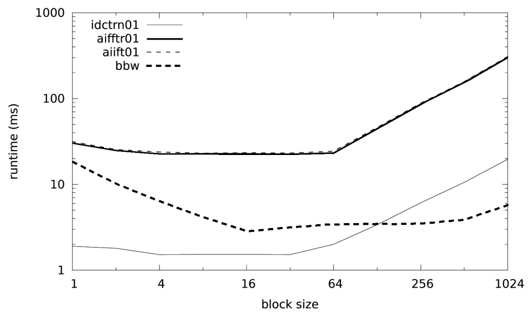

Figure 4: Effect of block size on kernel runtime

is that a warp is executed as a group of 32 threads. Thus, block sizes in the aforementioned range will require minimal or no context switches.

When a block contains a single thread, then for most benchmarks the execution time remains close to optimal, except for the BBW benchmark. A block of size one implies that none of 1024 threads are executing in lock-step. If a benchmark is frequently accessing global memory, then due to the absence of lock step execution the kernel runtime will increase. The BBW benchmark has such a memory access pattern. For each thread iteration in BBW, frequently accessed inputs to the thread are located in global memory. One may refactor the code to move inputs to the thread-local memory, but GPU cards only offer a very small amount of thread-local memory (few KBs). Thus, such code refactorings become nontrivial when the input data structures are large.

Data transfer between host and device.

A key component of our approach is to copy the inputs for a test suite from the CPU memory to the GPU memory. These transfers start to gain importance when the benchmark requires large inputs. Notice in Table 1 that the memory transfers between the CPU and the GPU take more time than kernel execution for large test runs on aiifft01 and aifftr01. While the issue pertaining to GPU-CPU memory transfer is not a show stopper for our benchmarks, it assumes much larger importance for benchmarks where the complete input may not fit in the GPU memory. In such cases, either GPU cards with larger memory have to be procured or the GPU code will involve nontrivial refactoring where the kernel operates on partial inputs at a time and synchronises with the CPU before operating on the rest. Consequently, transfer latency and bandwidth limitations of the PCI express link will become more apparent.

The recent road maps of companies developing GPU cards indicate that the next-generation GPUs try to address the problem of CPU-GPU memory transfers. Some of the solutions have already been released, including unified virtual memory and the fabrication of the GPU chip on the same die as the CPU (Kaveri [4]). We believe that with advances in GPU technology, test acceleration via GPUs will become only even more attractive.

System Calls.

Currently there are few abstractions available that allow GPU code to perform system calls (such as `brk()`, file I/O) and callbacks. With current state-of-the-art GPU technology, it is still nontrivial to run benchmarks that frequently perform system calls on GPUs. This is an active area of research [22, 17]. Our benchmarks notably did not have system calls.

4.4 Threats to Validity

The first threat to validity is the small number of programs used in our experiments. We have only used four programs, even if they are all industry standard programs. The second threat arises from the fact that we only use programs from the automotive domain. Programs from other domains were not used in our experiments. We plan to do a more extensive evaluation using programs from different application domains in the future.

5. CONCLUSION

In this paper, we proposed an approach to accelerate test execution using GPUs and explored its feasibility. Our approach inserts GPU device management code in the program interface to launch a GPU thread for each test case. The program functionality is not modified in our approach. We evaluated our approach using programs in the embedded systems domain – 3 benchmark programs from EEMBC and one brake-by-wire system from Volvo. We ran the programs on test suites with sizes that range from 1024 to 16384 tests. Our approach using GPUs achieved speedups in the range of 10 to 27 times for the EEMBC benchmarks, and a speedup of 2 for the brake-by-wire system. The extent of control flow divergence in a program affects the speedup achieved with GPUs. We verified, for the 4 benchmark programs, that our approach generated the same test case outputs over all the tests as the CPU. Finally, we also discussed limitations in GPUs and the restrictions they impose in the context of our approach.

Acknowledgements

This work was supported by the ARTEMIS VeTeSS project and ERC project 280053.

References

[1] “http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html”.

[2] “http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc”.

[3] "https://www.khronos.org/opencl/".

[4] “http://www.amd.com/en-gb/innovations/software-technologies/compute-cores”.

[5] “http://www.comp.dit.ie/rlawlor/alg_ds/sorting/quicksort.c”.

[6] "http://www.eembc.org/".

[7] “http://www.mathworks.co.uk/products/simulink/”.

[8] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *High Performance Distributed Computing*, pages 165–174. ACM, 2008.

[9] M. P. E. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *ASE*, pages 176–185, 2004.

[10] J.-H. Huang. “http://www.gputechconf.com/page/live-stream-source1.html”.

[11] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *IPDPS*, pages 1–8. IEEE, 2009.

[12] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar. Operating systems challenges for GPU resource management. In *Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2011.

[13] A. Lashgar and A. Baniasadi. Performance in GPU architectures: Potentials and distances. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2011.

[14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[15] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[16] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009.

[17] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Operating Systems Principles (SOSP)*, pages 233–248. ACM, 2011.

[18] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Code Generation and Optimization (CGO)*, pages 195–204. ACM, 2008.

[19] P. J. Schroeder and B. Korel. Black-box test reduction using input-output analysis. In *ISSTA*, pages 173–177. ACM, 2000.

[20] T. Scogland, H. Lin, and W. Feng. A first look at integrated GPUs for green high-performance computing. *Computer Science-Research and Development*, 25(3):125–134, 2010.

[21] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code. In *High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2010.

[22] J. A. Stuart, M. Cox, and J. D. Owens. GPU-to-CPU callbacks. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *LNCS*, pages 365–372. Springer, 2011.

[23] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification & Reliability*, 22(2):67–120, 2012.

[24] S. Yoo, M. Harman, and S. Ur. Highly scalable multi objective test suite minimisation using graphics cards. In *Search Based Software Engineering (SSBSE)*, pages 219–236. Springer, 2011.

[25] Z. Yu, J.-H. Cho, B.-W. Oh, and L.-S. Lee. Parallel algorithm for generation of test recommended path using cuda. *International Journal of Engineering Science & Technology*, 5(2), 2013.