CSC 258 Term Project: Feasibility of circuit programming models for GPUs

Vladimir Maksimovski, with much help from Sreepathi Pai

May 2020

1 Introduction

1.1 Graphics Processing Units

Discrete Graphics Processing Units (GPUs) are becoming increasingly more prevalent in the high-performance computing (HPC) community. They are specialized circuits optimized for running single-instruction, multiple-thread (SIMT) code, provided by the user as a kernel. These kernels can be programmed using a framework like CUDA or OpenCL, in a high-level language like C++. As discrete GPUs are distinct from the CPU, communication can only be done via the motherboard bus. As this is limited, GPUs generally have dedicated DRAM. For example, the NVIDIA GeForce 1080 has 8GB GDDR5X memory, with a theoretical peak bandwidth of 320GB/sec.

The user launches a kernel by specifying a size for the grid, which is composed of blocks, which are further subdivided into threads. Built-in synchronization primitives are usually available within the same block.

In this project, for specific details I will focus on the NVIDIA Geforce 1080, and write GPU code using CUDA.

GPUs have such a high number of compute elements that many programs can be memory-bound rather than compute-bound. Clearly, if a kernel saturates the memory, then this is a hardware limitation that cannot be avoided. It is not always possible to create a memory-saturated kernel implementation, but in many cases a good metric for kernel performance is kernel bandwidth divided by memory bandwidth. Kernel bandwidth is calculated using the formula:

$$size/exec_seconds,$$

where $size$ is the memory size of all arrays, while $exec_seconds$ stores the execution time in seconds. The result metric is in Gigabits per second.

A high-level description of a CUDA program looks as follows:

- The user allocates GPU memory for the data.
- The user transfers the motherboard data to the GPU memory.
- The user launches the kernel (possible multiple different kernels)
- The user copies the result data to the motherboard memory.

An example CUDA program implementing vector addition is shown in Fig. 1.

```cpp
__global__ void add(int n, float *x, float *y, float *ans){
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if(i < n){
        ans[i] = x[i] + y[i];
    }
}

// x, y are pointers to vectors of size n.
float* vector_addition_using_gpu(int n, float *x, float *y){
    float *ans = (float *) malloc(n * sizeof(float));
    assert(ans != NULL);

    float *d_x, *d_y, *d_ans;
    cudaMalloc(&d_x, n * sizeof(float)); assert(d_x != NULL);
    cudaMalloc(&d_y, n * sizeof(float)); assert(d_y != NULL);
    cudaMalloc(&d_ans, n * sizeof(float)); assert(d_ans != NULL);

    cudaMemcpy(d_x, x, n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, n * sizeof(float), cudaMemcpyHostToDevice);

    int grid_size = (n + 256 - 1) / 256;
    add<<<grid_size, 256>>>(n, d_x, d_y, d_ans);

    cudaMemcpy(ans, d_ans, n * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_ans);

    return ans;
}
```

Figure 1: CUDA program implementing vector addition

1.2 Prefix Scan

An inclusive prefix scan algorithm receives an n-sized array a, and a binary associative operation $\oplus$, and returns an n-sized array b such that

$$b_i = a_1 \oplus a_2 \oplus \ldots \oplus a_i.$$

While computing b seems inherently sequential, this operation can be easily modeled into a parallel programming format.

A distinction is made between inclusive and exclusive prefix scans. An inclusive prefix scan calculates b_i as

$$a_1 \oplus a_2 \oplus \ldots \oplus a_i,$$

while for an exclusive prefix scan:

$$a_1 \oplus a_2 \oplus \ldots \oplus a_{i-1},$$

where b_1 stores the identity for $\oplus$.

Neither variant is more difficult to implement in parallel programming. Throughout the paper, I'll work with the inclusive prefix scan, as it doesn't require an identity element, which does not exist for certain operations, like maximum.

Throughout the project, I'll focus on this problem, as it's a non-trivial parallel programming problem, with many implementations available, each with different performance depending on the underlying GPU architecture.

1.3 Circuit programming model

As of 2020, CUDA supports C, C++ and Fortran. All three languages have been designed with single-instruction execution in mind. Internally, these get compiled down to PTX assembly, which is about as easy to use as x86 assembly. No matter what, the user is forced to write code in a model unnatural to the SIMT execution model.

Another model was that of a "circuit" programming model: where the user describes the kernel by describing the operations as a circuit over one block of threads. This is best suited for kernels with "static" memory access patterns, such as bitonic sorting, reduction, or prefix scan. Examples of programs which are difficult to express in this model would be sparse matrix-vector multiplication, or merging sorted lists into one sorted list.

This can be thought of as a computational graph, with a set of input lines and output lines representing the inputs and outputs to the network. An example of a prefix scan is shown in Fig. 2.

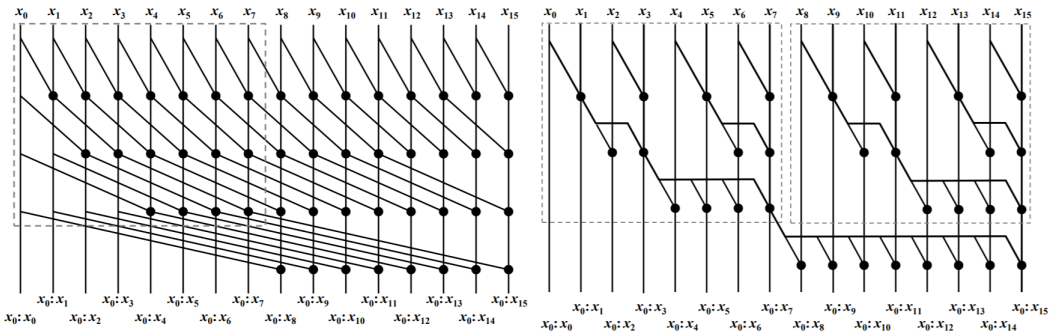

Figure 2: a) Kogge-Stone/ b) Sklansky circuits shown on the left and right respectively. The dotted lines show recursive pieces of the circuit. Figures are sourced from [2].

The most natural representation for the circuits would be a recursive one. Functional programming languages, such as Lisp, Haskell or Coq seem best suited for this purpose, as creating inductive patterns is natural in those languages.

This programming model, despite seeming simplistic, can be used to implement many commonplace GPU operations with a static access pattern, such as prefix-scan, reduce, vector-addition, vector-multiplication, etc.

Strengths of this model is that it can be visualized easily, and expresses the kernel at a higher level of abstraction than a C++ program, allowing the compiler to reason about the program and optimize better.

Of course, expecting a circuit compiler to write perfectly-optimized code would be impossible. As a design decision, this program will output executable CUDA code, which the user can refine to their liking. This means allow the developer to have a working and correct version of the circuit, improving developer productivity.

Due to scope, I will only focus on a different representation for the circuit, where the user specifies a circuit with a fixed number of inputs and outputs. The circuit is described as containing multiple stages, each separated with a synchronization barrier. In each stage, a thread executes an operation using multiple source lines, and writes to an output line. The user can assume that no data races

occur within each stage, and that all writes in one stage will be available in the next stage. I call this format the *CRC* format. Further documentation for this format is available in the appendix.

This format, while limited, can still be visualized easily. I have not written a program that visualizes these circuits, as this is more of an exercise in graphics programming than parallel programming.

2 Methodology

To examine the feasibility of this approach, I examined two things:

1. I created a python module which reads in a circuit in the CRC format, and outputs an executable C++ kernel.

2. I wrote multiple implementations of prefix scan algorithms. I further optimized one particular variant and analysed the performance benefits. In the evaluation section, I will consider each optimization's ability to be automated i.e. done by a compiler.

2.1 CRC to CUDA compiler

This project submission contains two relevant files: `proof_of_concept/circuit.py`, and `proof_of_concept/circuit2.py`. These take in a circuit represented in the CRC format, and output CUDA C++ code for the kernel.

The current implementation technique is very simple: each stage consists of a switch statement branching using the thread id, with a barrier between each stage. To avoid data races, multiple copies of the array are used. `circuit.py` uses many copies of the array (as many as the number of stages + 1), while `circuit2.py` only uses two copies of the array. I included `circuit.py` as it's simpler, however, `circuit2.py` is the recommended variant to use.

Also, as it's expected that the user will make changes to the kernel themselves, the kernel is output with normal C++ indentation (K&R style).

As a test, I created a simple reduction circuit with 8 inputs, and stored it at `proof_of_concept/reduce.cu`. It is shown visually in Fig. 3. The corresponding CUDA kernel is shown in Fig. 4.

The limitations of this method are easy to see. It is difficult to achieve any level of performance with this method, as each thread executes a different line of code, which is the opposite of the SIMT execution GPUs are optimized for. This would be alleviated by using a different representation for the circuit i.e. that of a recursive description.

Also, the current implementation only assumes one thread block. This would not be fixed by merely using a recursive description. Many CUDA programs work on a contiguous range of data, resulting in a uniform partitioning such as thread block 0 working on 0...15, thread block 1 works on 16...31, etc. The compiler could handle this by using different offset modes. This would mean that instead of using `x0[threadid]`, `x0[offset(blockid) + threadid]` would be used instead, where offset is a macro customizable by the user.

If in one stage only a few data lines change, the whole array will still be copied over to the next stage's array. This is very wasteful, but could be optimized in future work.

All in all, I believe that the copmiler works as a proof of concept, and sheds some light on the major areas of work to be done in the compiler.

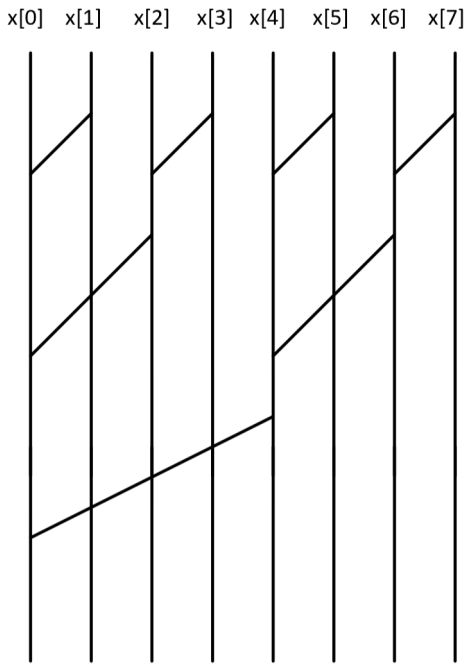

Figure 3: Reduction circuit described in sum.cu

2.2 Prefix scan algorithm implementations and optimizations

2.2.1 Kogge-stone prefix-scan

The Kogge-Stone algorithm for prefix scan is interesting, as its depth is optimal: O(log n) for n elements. It assumes that there are as many processors as the number of elements. However, it is possible to side-step this using a "double-buffering" technique to hide non-concurrent modifications.

It consists of log(n) stages. In the k-th stage, element x_i accumulates the value of x_{i-2^k} if $i - 2^k \geq 0$.

To avoid data races, two copies of the array are used, and they get switched between each stage.

An implementation of a Kogge-stone prefix-scan algorithm is available at `prefix_scan/naive_parallel.cu`.

This technique requires many GPU memory accesses, so it's far from optimal. However, the GPU can calculate this prefix scan at a bandwidth of 3.70 GBPS, which is still twice quicker than the sequential variant (1.88 GBPS).

2.3 Using more than one kernel

In the previous algorithm, all memory accesses were to GPU DRAM memory. As accessing GPU DRAM has a high latency, it is usually much better to copy the relevant elements to shared memory, compute the prefix scan over them, and then write them back to GPU DRAM. The next variant uses this technique.

```
__global__ void sum(int *x0, int *x1){
    int tid = threadIdx.x;
    x1[tid] = x0[tid];
    switch(tid){
        case 0:
            x1[0] += x0[1];
            break;
        case 2:
            x1[2] += x0[3];
            break;
        case 4:
            x1[4] += x0[5];
            break;
        case 6:
            x1[6] += x0[7];
            break;
    }
    __syncthreads();
    x0[tid] = x1[tid];
    switch(tid){
        case 0:
            x0[0] += x1[2];
            break;
        case 4:
            x0[4] += x1[6];
            break;
    }
    __syncthreads();
    x1[tid] = x0[tid];
    switch(tid){
        case 0:
            x1[0] += x0[4];
            break;
    }
    __syncthreads();
    x0[tid] = x1[tid];
    switch(tid){
    }
}
```

Figure 4: Reduction kernel created using the CRC to CUDA compiler.

2.4 Work-efficient prefix-scan

2.4.1 Algorithm description

In GPUs there's a notion of "work-efficiency". This is the total number of operations needed for all processors. For the Kogge-stone prefix scan, work totals $O(n \cdot logn)$. In the sequential code, clearly only $O(n)$ work is being done. There's a parallel algorithm implementing prefix-scan which is work-efficient.

We first calculate the prefix scan of all threads in the same thread-block. We copy the elements from GPU memory to shared memory for better latency, and write the results to GPU memory later.

The idea is to use two stages: an upsweep and a downsweep stage. The upsweep stage consists of multiple recursive levels. At each level, every even element is accumulated with each predecessor. In the next stage, only even elements are considered, so w.r.t the whole array every fourth element is accumulated with the second predecessor.

For an eight element array this means the first, third, fifth and seventh elements store a single element, the second and sixth elements store the sum of the previous two elements, and the fourth element stores the sum of the first four elements.

Notice that the sixth element stores only the sum of the fifth and sixth elements, while it should contain the sum of the first through sixth elements. This gets corrected in the downsweep stage.

The downsweep stage consists of $logn$ iterations. No matter how I phrase it, Fig. 5 makes the idea much clearer. The code for this involves hard-to-explain bit tricks.

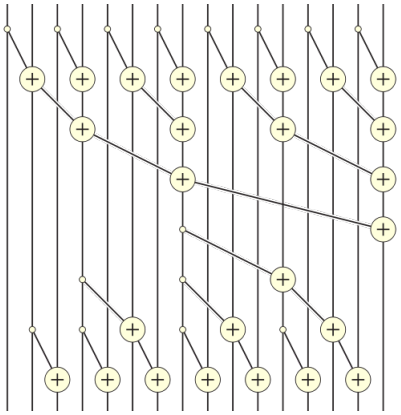

Figure 5: Work-efficient prefix-scan circuit. Sourced from Wikipedia article on prefix sums[1].

After the two stages, we have the within-thread-block prefix scan of each element. Since thread blocks can't communicate with one another, we need a technique that would take the sum of a thread block and propagate it to successive thread blocks. This is done using a different kernel.

The sum of each thread block is stored to another array. Then, we calculate the prefix-scan of this array. We again store the sum of each thread block to another array, resulting in an even smaller array. We proceed to do this until we get a smaller array than the number of threads in a block.

We then go over these same arrays in reverse order, from smallest to largest. In each step, we propagate the changes using the `propagate` kernel from the smaller array's thread-block-scans to the larger array's thread-scans. After doing this, we have the corrent prefix-scan of each element in the array.

The performance of this technique is much better: 34.40 GBPS bandwidth. This shows that different circuits for the same problem can have different performance, so allowing developers to experiment with them more easily will result in much better productivity.

The above algorithm is implemented in `prefix_scan/parallel_main.cu`. I use 128 threads per block.

2.4.2 Stride optimization

All threads in the prefix scan kernel did some work: they copied a part of the input array to shared memory. However, during the upsweep and downsweep stages, only a fraction of elements did any work. During upsweep, $n/2$ threads do work in the first iteration, $n/4$ in the second iteration etc. Worst of all, threads doing work and threads not doing work may be adjacent, resulting in divergence. This is very wasteful, as GPUs are optimized for SIMT executions. To fix this, I applied a few optimizations to alleviate these issues.

- Instead of using 128 threads per block, I use 64 threads. Each thread copies two elements of the input array to shared memory at the start of the kernel, and back to the input array at the end of the kernel.
- Threads doing work are now grouped together. In the original implementation, the first iteration of upsweep involved the second, fourth, sixth, 128-th threads. In the current implementation, the first, second, third, ..., 64-th are used, reducing divergence. This is done by having the first thread "substitute" for the second thread, the second thread "substituting" for the fourth thread etc.

The improvements result in a bandwidth of 42.20 GBPS. The code is stored in `prefix_scan/parallel_strided.cu`.

2.4.3 Reverse optimization

In the strided optimization variant, in the upsweep and downsweep stages, each thread needs to calculate an "actual thread id", and branch whether it is active in the current iteration based on it. Calculating the new thread id takes some work, which may be thrown out depending on the branch.

An observation is that in the first upsweep iteration, only $n/2$ threads do work, $n/4$ in the second iteration etc. So a simple comparison between *threadid* and $n/2^i$ would save some work.

The improvements from this change are very small: 42.79 GBPS bandwidth. The code is stored in
`prefix_scan/parallel_reversed.cu`.

2.4.4 Bank conflict optimization

In the Pascal architecture, the shared memory is partitioned in 32 banks. Memory addresses are distributed according to the lowest 5 bits: memory address 000000 belongs to bank 0, memory address 000001 belongs to banks 1. This wraps around, so memory address 100000 belongs to bank 0. Accesses to the same bank by different thread blocks results are serialized, so they should be avoided. Accesses to the same bank by threads in the same block, however, are broadcast once, resulting in no performance drawback.

In the prefix scan kernel, in the fifth iteration of the upsweep stage, every 32nd element is being accessed, which results in many bank conflicts.

To fix this, I applied the following optimization:

- I added an offset to each shared memory address. Now, memory address 000000 belongs to bank 0, memory address 100000 belongs to bank 1, memory address 1000000 belongs to bank 2 etc.

After this optimization, memory bandwidth was at 50.23 GBPS. This is the best optimization I could achieve. Results are stored at `prefix_scan/parallel_banks.cu`.

3 Discussion

Here, I'll evaluate each topic in the previous section with regards to what it can tell us about the circuit compiler.

3.1 CRC to CUDA compiler

The current condition of the compiler, and the difficulties described show that the current limited representation will not work. It is also very difficult to separate issues arising from the current limited representation, and those arising from the overall idea. The only way to resolve these concerns is by researching into the subject.

Currently, the best proposal would be to use a inductive representation of the grid. In this, the user would specify the base case representation (n=1), and a inductive representation, where the circuit contains one or more self-similar parts of smaller size, along with some combining operations.

Consider Fig. 2 b). Clearly, a single circuit line by itself is the base case. The inductive step is: take two circuits of equal size, and then connect the last line of the first circuit with all lines of the second circuit. This is a very compact representation.

While this is also possible for Fig. 2 a), I instead chose to focus on b), as the recursion was easier to explain.

In order to get the compiler to output (decently) performant CUDA C++ kernels, much work will need to be done. I believe that using a more natural representation (recursion) will result in surprisingly non-trivial optimizations from the compiler, such as the "striding" optimization shown in Section 2.2.

3.2 Prefix scan optimization feasibility

3.2.1 Stride optimization

Consider the original `parallel_main.cu` prefix scan kernel. Figuring out the strided optimization resulted in a very non-trivial modification of the code, and resulted in quite a performance benefit.

However, from the circuit description, having a strided pattern is apparent. I believe that a compiler could feasibly create this optimization.

3.2.2 Reverse optimization

The reverse optimization did not bring much benefit, so I will not focus on it.

3.2.3 Bank conflict optimization

The bank conflicts optimization is very interesting. In required some thought from the developer in how shared memory is organized, how memory is distributed to the banks, what the memory access pattern is like, and what the proper offset should be. I think this is a very intricate optimization, which would be difficult for a compiler to do without large degrees of hard-coded behavior.

I do not believe this optimization will be easily feasible for a circuit compiler.

4 Future work

A lot of the work done is this project was caveated by the bad choice of representation. Currently, my best idea would be to use a inductive definition expressed in some functional language, like Haskell.

5 Conclusion

Say that the current proof-of-concept shows that a lot of work needs to be done. The change of representation from the actual circuit to a recursive format should help out a lot. It seems like some very important optimizations such as the stride optimization are feasible in such programs. I will plan on taking CSC 254: Programming Language Design and Implementation, a course which will teach me about compiler architecture and optimizations. I had trouble finding GPU programming models different from assembly/C/C++/Fortran, so I believe the overall goal of representing GPU programs more naturally is a worthwhile pursuit.

6 Appendix I: CRC format description

The CRC format describes a circuit. A visualization of one such circuit is shown in Fig. 3. An example program is available in `proof_of_concept/sum.crc`.
In the first line of the CRC file, the user describes the name of the kernel, the number of input lines, the number of stages, and the name of the array.
Then, the list of operations follows. an operation is described in terms of the stage number, the result line, the operation, and then the input line.
The python script skips empty lines, and lines containing # symbols. # symbols are intended to denote comments.

7 Appendix II: Documentation

- Report.pdf - currently open document.
- vector_addition - Contains simple code for vector addition in CPU and CUDA.
  - add.cpp - Example CPU implementation of vector addition for comparison.
  - add.cu - Contains multiple CUDA implementations of vector addition. It chooses which algorithm to use using an argument 1-4.
  - test.sh - Script to test out add.cpp and add.cu
  - results.txt - Execution log of test.sh
  - reader.py - Ignore this file; it's only there to make test.sh work.
- prefix_scan - Contains code for prefix scans in CPU and CUDA.
  - prefix_scan.cpp - Sample implementation of CPU prefix scan for comparison.
  - naive_parallel.cu - Kogge stone prefix scan.
  - parallel_main.cu - Work-efficient parallel scan.
  - parallel_strided.cu - Strided implementation of parallel scan.
  - parallel_reversed.cu - Optimized implementation of parallel scan.
  - parallel_banks.cu - Bank-conflict-optimized implementation of parallel scan.
  - test.sh - Script which runs all the prefix scan files, and computes their bandwidth.
  - results.txt - Execution log of test.sh.
  - reader.py - Ignore this file; it's only there to make test.sh work.
- proof_of_concept - Contains code which compiles from CRC to CUDA kernels.
  - circuit.py - Simple implementation of CRC to CUDA converter.
  - circuit2.py - CRC to CUDA compiler which uses only 2*n memory.
  - sum.crc - CRC description of a reduction kernel.
  - sum.cu - Executable CUDA code containing the compiled CRC kernel from sum.crc.
  - prefix_sum.crc - CRC description of a Kogge-Stone adder.
  - prefix_sum.cu - Executable CUDA code containing the compiled CRC kernel from prefix_sum.crc

8 Appendix III: Reproducibility

I've talked a lot in this project. I've also added some source files, so that experiments can be done at the reader's desire.
The directory vector_addition contains a test.sh file. This can be executed on the gpu-node01.csug.rochester.edu to obtain results for performance of the vector addition kernels. Just in case, I included an example execution in results.txt
The directory prefix_scan similarly contains a test.sh file, which is also expected to be run on gpu-node01.csug.rochester.edu. Example execution is available in results.txt
The directory proof_of_concept does not contain such an easily executable file. The reader should try the commands
`python3 circuit2.py sum.crc`,
`python3 circuit2.py prefix_sum.crc`,
and the output of this should be compared with the kernels at `sum.cu`, `prefix_sum.cu`. The reader can compile and execute the .cu files in the directory, and see that they work as the circuit describes them.

9 References

[1] https://en.wikipedia.org/wiki/Prefix_sum

[2] https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf