Australian National University

Atomics

Dr Eric McCreath

Research School of Computer Science

The Australian National University

Timing

A very simple way of timing how long a thread, or part of a thread, takes to execute is use the "clock" function within the kernel.

So you could time a kernel something like:

```
__global__ kernel (int *timing, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        clock_t start = clock();
        // do your work
        clock_t stop = clock();
        timing[idx] = stop - start;
    }
}
```

This gives you a simple way of targeting the timing of the key parts of your code. Once you have transferred "timing" back to the host you can do an evaluation of average, min, max, and variance across the threads.

Atomics

Within threads you can directly access shared or global memory. In general each thread should read and write to its own part of the memory. If threads are just reading common parts of memory within a kernel execution then the program will work correctly. The problem arises when multiple threads write/read to the same memory locations. If the operation you wish to do on common memory are simple enough then a very easy way of address this is using atomic operations. They include:

```
atomicAdd() - adds to a value  (can be integer or floating point)
atomicSub() - subtracts from a value
atomicExch() - does an atomic exchange of a memory
atomicMin() - replace with the minimum of current and provided value
atomicMax() - replace with the maximum of current and provided value
atomicInc() - increment the value of a memory location
atomicDec() - decement the value of a memory location
atomicCAS() - compare and swap
atomicAnd() - bitwise "and" operator
atomicOr() -  bitwise "or" operator
atomicXor() -  bitwise "xor" operator
```

atomicAdd

The atomicAdd operator works on a variety of sized integers (16 bit, 32 bit and 64 bit), although this is a little bit dependent on which version of Cuda you are using. There is also 16 bit, 32 bit and 64 bit float versions of this operator. The 32 bit integer version has the signature:

```
int atomicAdd(address, value);
```

Basically it loads the integer located at "address" adds "value" to it and stores this sum back to "address". The function returns the "address" value prior to it being added to. The below code sums an array, although it would work correctly it would perform poorly.

```
__device__ int sum = 0;
__global__ sumArray(int *data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        atomicAdd(&sum,data[idx]);
    }
}
```

atomicInc

"atomicInc" works on unsigned 32 bit integers. What is interesting about this atomic is it wraps around effectively providing "modulo" increment. The signature is:

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
```

The value at "address" is incremented with the result stored back into "address". The function returns the value of "address" before it has been incremented. The increment works modulo "val", so the "address" is never set to "val" rather it wraps back to 0.

You could use this function for count the number of situations that occur occasionally. Also if only some of your threads need to store a result this operator provides a simple way of working out the location to store it.

```
if (storeResult) {
    int pos = atomicInc(resultPos, 4294967295);
    if (pos < resultArraySize) resultArray[pos] = someResult;
}
```

Locks!!!

In theory you could use "atomicExch" or "atomicCAS" to implement traditional locks or semaphores. However this is best avoided as when you think of the SIMT model of execution. Basically if you had a lock in your code all the threads of the warp will hit the lock statement at the same time creating a huge amount of divergence.

Divergence

With the SIMT approach the threads within a warp can take different paths through the code, however, as it is still "Single Instruction" only one parth is executed at any one time. While one path is executed the threads that follow different paths are masked. That is until the paths converge or the alternative paths are followed. This splitting of the paths different threads follow is known as divergence.

```
int lane = idx % 32;
if (lane < 16) {
    doPart1();
} else {
    doPart2();
}
```

Thread instruction mask

11111111111111111111111111111111

11111111111111111111111111111111

11111111111111110000000000000000

00000000000000001111111111111111

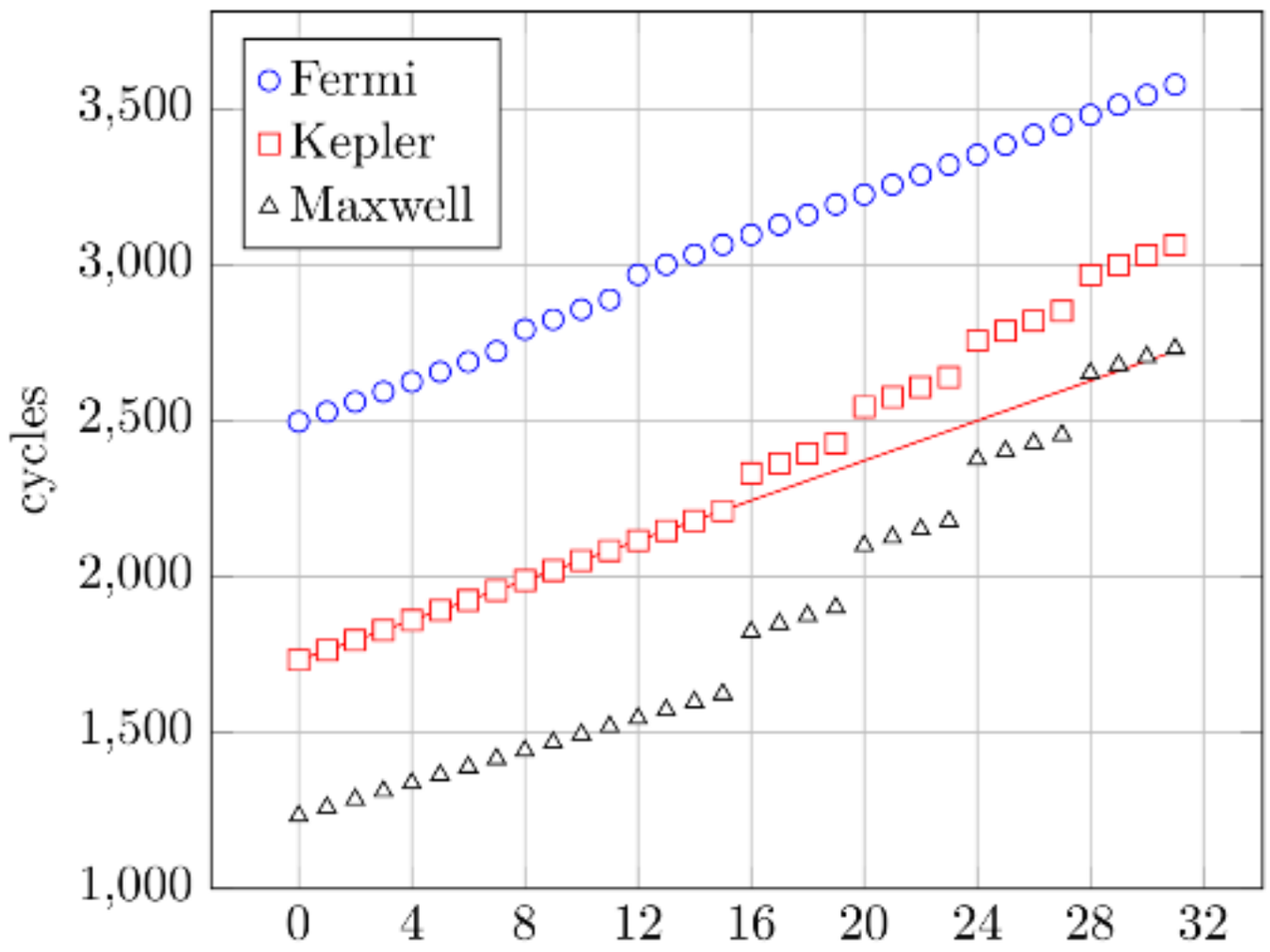
11111111111111111111111111111111

Divergence

Divergence can significantly effect performance. So it is worth minimizing if possible.

The below graph is from "Benchmarking the cost of thread divergence in CUDA" P. Bialas and A. Strzelecki, 2015

Time for an increasingly divergent loop



https://arxiv.org/pdf/1504.01650.pdf

References

- Cuda Quick Reference Card

http://www.info.univ-angers.fr/pub/richer/cuda/CUDA_C_QuickRef.pdf

- Cuda c Programming Guide

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html