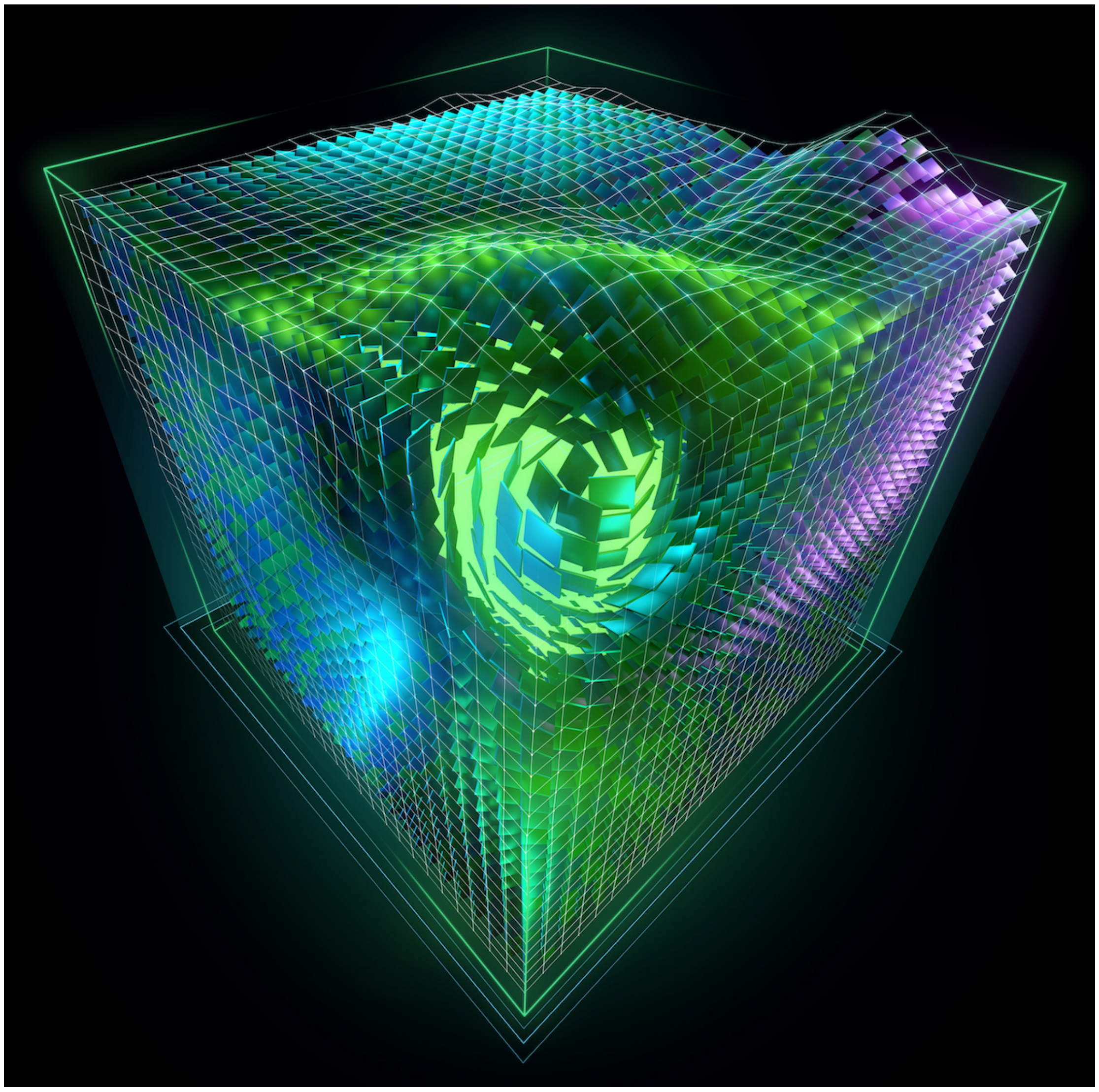
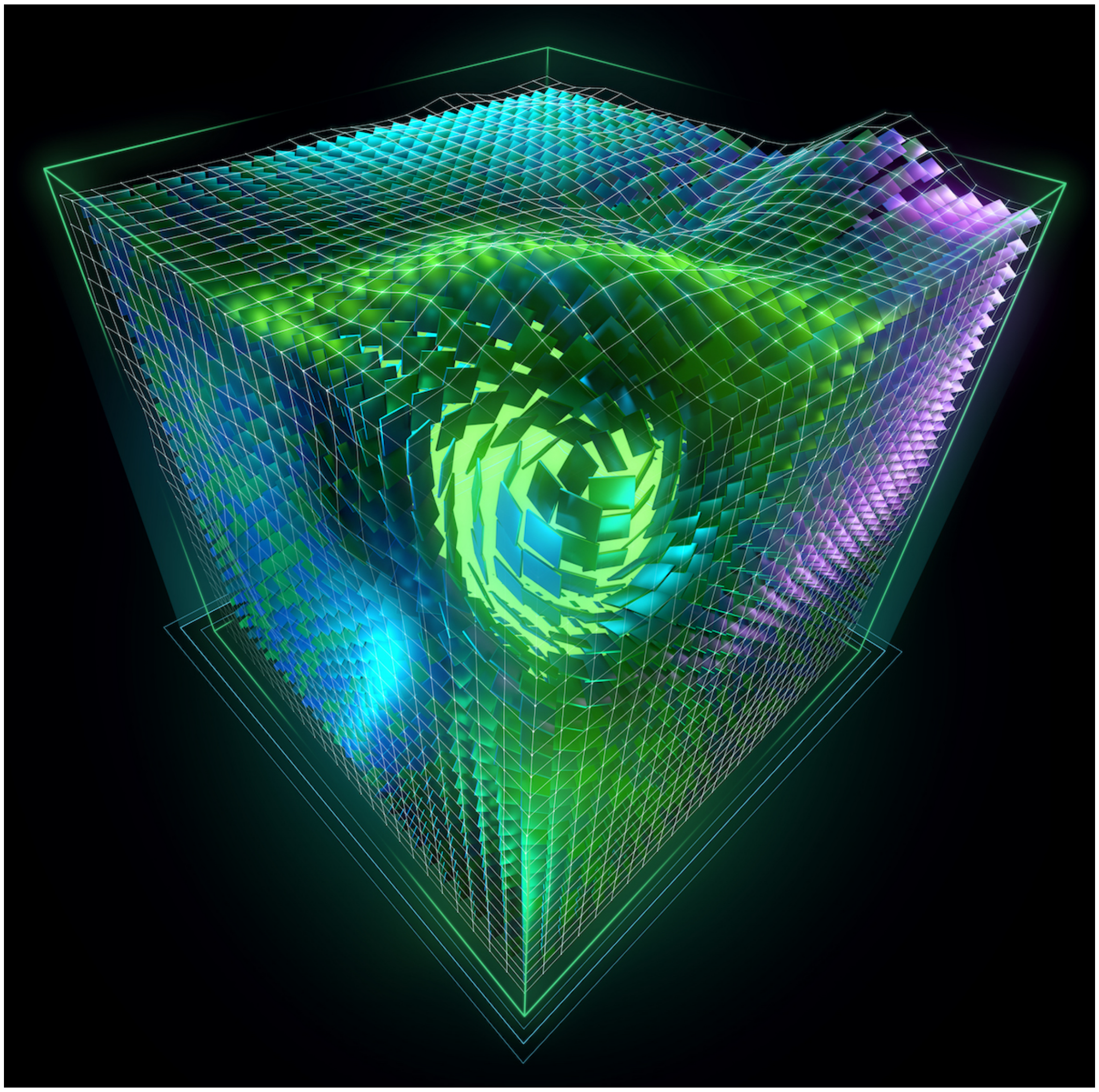
How to Implement Performance Metrics in CUDA C/C++





In the first post of this series we looked at the basic elements of CUDA C/C++ by examining a CUDA C/C++ implementation of SAXPY. In this second post we discuss how to analyze the performance of this and other CUDA C/C++ codes. We will rely on these performance measurement techniques in future posts where performance optimization will be increasingly important.

CUDA performance measurement is most commonly done from host code, and can be implemented using either CPU timers or CUDA-specific timers. Before we jump into these performance measurement techniques, we need to discuss how to synchronize execution between the host and device.

Host-Device Synchronization

Let’s take a look at the data transfers and kernel launch of the SAXPY host code from the previous post:

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

saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

The data transfers between the host and device using cudaMemcpy() are *synchronous* (or *blocking*) transfers. Synchronous data transfers do not begin until all previously issued CUDA calls have completed, and subsequent CUDA calls cannot begin until the synchronous transfer has completed. Therefore the `saxpy` kernel launch on the third line will not issue until the transfer from `y` to `d_y` on the second line has completed. Kernel launches, on the other hand, are asynchronous. Once the kernel is launched on the third line, control returns immediately to the CPU and does not wait for the kernel to complete. While this might seem to set up a race condition for the device-to-host data transfer in the last line, the blocking nature of the data transfer ensures that the kernel completes before the transfer begins.

Timing Kernel Execution with CPU Timers

Now let’s take a look at how to time the kernel execution using a CPU timer.

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

t1 = myCPUTimer();
```

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

In addition to the two calls to the generic host time-stamp function `myCPUTimer()`, we use the explicit synchronization barrier `cudaDeviceSynchronize()` to block CPU execution until all previously issued commands on the device have completed. Without this barrier, this code would measure the kernel *launch time* and not the kernel *execution time*.

Timing using CUDA Events

A problem with using host-device synchronization points, such as `cudaDeviceSynchronize()`, is that they stall the GPU pipeline. For this reason, CUDA offers a relatively light-weight alternative to CPU timers via the CUDA event API. The CUDA event API includes calls to create and destroy events, record events, and compute the elapsed time in milliseconds between two recorded events.

CUDA events make use of the concept of CUDA *streams*. A CUDA stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped—a property that can be used to hide data transfers between the host and the device (we will discuss this in detail later). Up to now, all operations on the GPU have occurred in the default stream, or stream 0 (also called the “Null Stream”).

In the following listing we apply CUDA events to our SAXPY code.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

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

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

CUDA events are of type cudaEvent_t and are created and destroyed with cudaEventCreate() and cudaEventDestroy(). In the above code cudaEventRecord() places the start and stop events into the default stream, stream 0. The device will record a time stamp for the event when it reaches that event in the stream. The function cudaEventSynchronize() blocks CPU execution until the specified event is recorded. The cudaEventElapsedTime() function returns in the first argument the number of milliseconds time elapsed between the recording of start and stop. This value has a resolution of approximately one half microsecond.

Memory Bandwidth

Now that we have a means of accurately timing kernel execution, we will use it to calculate bandwidth. When evaluating bandwidth efficiency, we use both the theoretical peak bandwidth and the observed or effective memory bandwidth.

Theoretical Bandwidth

Theoretical bandwidth can be calculated using hardware specifications available in the product literature. For example, the NVIDIA Tesla M2050 GPU uses DDR (double data rate) RAM with a memory clock rate of 1,546 MHz and a 384-bit wide memory interface. Using these data items, the peak theoretical memory bandwidth of the NVIDIA Tesla M2050 is 148 GB/sec, as computed in the following.

$BW_{\text{Theoretical}} = 1546 * 10^6 * (384/8) * 2 / 10^9 = 148$ GB/s

In this calculation, we convert the memory clock rate to Hz, multiply it by the interface width (divided by 8, to convert bits to bytes) and multiply by 2 due to the double data rate. Finally, we divide by 10^9 to convert the result to GB/s.

Effective Bandwidth

We calculate effective bandwidth by timing specific program activities and by knowing how our program accesses data. We use the following equation.

$BW_{\text{Effective}} = (R_B + W_B) / (t * 10^9)$

Here, $BW_{\text{Effective}}$ is the effective bandwidth in units of GB/s, R_B is the number of bytes read per kernel, W_B is the number of bytes written per kernel, and t is the elapsed time given in seconds. We can modify our SAXPY example to calculate the effective bandwidth. The complete code follows.

```
#include

__global__
```

```
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 20 * (1 << 20);
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

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

  cudaEventRecord(start);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y);

  cudaEventRecord(stop);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  cudaEventSynchronize(stop);
  float milliseconds = 0;
  cudaEventElapsedTime(&milliseconds, start, stop);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = max(maxError, abs(y[i]-4.0f));
  }

  printf("Max error: %fn", maxError);
  printf("Effective Bandwidth (GB/s): %fn", N*4*3/milliseconds/1e6);
}
```

In the bandwidth calculation, `N*4` is the number of bytes transferred per array read or write, and the factor of three represents the reading of `x` and the reading and writing of `y`. The elapsed time is stored in the variable `milliseconds` to make units clear. Note that in addition to adding the functionality needed for the bandwidth calculation, we have also changed the array size and the thread-block size. Compiling and running this code on a Tesla M2050 we have:

```
$ ./saxpy
Max error: 0.000000
Effective Bandwidth (GB/s): 110.374872
```

Measuring Computational Throughput

We just demonstrated how to measure bandwidth, which is a measure of data throughput. Another metric very important to performance is computational throughput. A common measure of computational throughput is GFLOP/s, which stands for “Giga-FLoating-point OPerations per second”, where *Giga* is that prefix for 10^9. For our SAXPY computation, measuring effective throughput is simple: each SAXPY element does a multiply-add operation, which is typically measured as two FLOPs, so we have

$$GFLOP/s_{\text{Effective}} = 2N / (t * 10^9)$$

N is the number of elements in our SAXPY operation, and t is the elapsed time in seconds. Like theoretical peak bandwidth, theoretical peak GFLOP/s can be gleaned from the product literature (but calculating it can be a bit tricky because it is very architecture-dependent). For example, the Tesla M2050 GPU has a theoretical peak single-precision floating point throughput of 1030 GFLOP/s, and a theoretical peak double-precision throughput of 515 GFLOP/s.

SAXPY reads 12 bytes per element computed, but performs only a single multiply-add instruction (2 FLOPs), so it's pretty clear that it will be bandwidth bound, and so in this case (in fact in many cases), bandwidth is the most important metric to measure and optimize. In more sophisticated computations, measuring performance at the level of FLOPs can be very difficult. Therefore it's more common to use profiling tools to get an idea of whether computational throughput is a bottleneck. Applications often provide throughput metrics that are problem-specific (rather than architecture specific) and therefore more useful to the user. For example, "Billion Interactions per Second" for astronomical n-body problems, or "nanoseconds per day" for molecular dynamic simulations.

Summary

This post described how to time kernel execution using the CUDA event API. CUDA events use the GPU timer and therefore avoid the problems associated with host-device synchronization. We presented the effective bandwidth and computational throughput performance metrics, and we implemented effective bandwidth in the SAXPY kernel. A large percentage of kernels are memory bandwidth bound, so calculation of the effective bandwidth is a good first step in performance optimization. In a future post we will discuss how to determine which factor—bandwidth, instructions, or latency—is the limiting factor in performance.

CUDA events can also be used to determine the data transfer rate between host and device, by recording events on either side of the `cudaMemcpy()` calls.

If you run the code from this post on a smaller GPU, you may get an error message regarding insufficient device memory unless you reduce the array sizes. In fact, our example code so far has not bothered to check for run-time errors. In the next post, we will learn how to perform error handling in CUDA C/C++ and how to query the present devices to determine their available resources, so that we can write much more robust code.