GPU Computing with CUDA

Hands-on: CUDA Profiling, Thrust

Dan Melanz & Andrew Seidl

Simulation-Based Engineering Lab Wisconsin Applied Computing Center Department of Mechanical Engineering University of Wisconsin-Madison



CUDA Profiling

*dct8x8.vp 23				- 0	Properties 23 Detail Graph	
	161.7 ms	161.8 ms	161.9 ms	162 ms	CUDAkernel1DCT/fleats int	int int)
Process: 11119	10117 113	101.0 110	LOLIPING	101,110	CODARemeriDer(noat-, int,	inc, inc)
Thread: -1494415584					Name	Value
Runtime API				cudaMemcpv2D	Start	161.329 ms
Driver API					Duration	106.132 µs
[0] GeForce GTX 480					Grid Size	[64,64,1]
Context 1 (CUDA)					Block Size	[8,8,1]
T MemCpy (HtoD)					Registers/Thread	14
T MemCpy (DtoH)			Me	mcpy DtoH [sync]	Shared Memory/Block	512 bytes
T MemCpy (DtoD)					- Memony	
Compute	CUDAkernelQua	CUDAkernel1IDCT(floa	t*, int		Clobal Load Efficiency	n/n
9 0.7% [101] CUD					Clobal Chara Efficiency	Tya
Y 0.3% [10] CUDAk					Global Store Efficiency	100%
W 0.0% [2] CUDAke	CUDAkernelQua				DRAM Utilization	10.9% (18.4
9 0.0% [1] CUDAke		CUDAkernel1IDCT(floa	t*, int		 Instruction 	
🍸 0.0% [1] CUDAke					Branch Divergence Overhe	0%
W 0.0% [1] CUDAke					Total Replay Overhead	§ 51%
¥ 0.0% [1] CUDAke					Shared Memory Replay Ov	0%
\[0.0% [1] CUDAke					Global Memory Benlay Ove	4 51%
E Streams					Global Cache Benjau Querk	016
Stream 1	CUDAkernelQua	CUDAkernel1IDCT(floa	t*, int Me	mcpy DtoH [sync]	Global Cache Replay Over	070
					Local Cache Replay Overne	076
	1				 Occupancy 	
Analysis 🕮 🖬 Details 🖷	Console Settings	alvsis Results				
Reset All	alyze All	High Branch Divergence O	erhead [35 1% avg	for kernels accountin	a for 1.9% of compute 1	
Timeline	o	Divergent branches are causing	significant instruction is	sue overhead.	g for 15% of compute j	More.
Multiprocessor	O 4	High Instruction Replay Ov A combination of global, shared	erhead [46.6% avg, f I, and local memory repla	or kernels accounting and causing signification	g for 39.1% of compute] nt instruction issue overhead.	More.
Kernel Memory	ø .	High Global Memory Instru	ction Replay Overhea	d [45.9% avg, for ke	rnels accounting for 39.1% of	compute]
Kernel Instruction		Non-coatesced global memory	accesses are causing sig	nificant instruction issue	overnead.	More.
Action action	•					







CUDA Code Profiling



 We will be using the CUDA Visual Profiler to profile a matrix addition problem:

$\mathbf{C} = \mathbf{A} + \mathbf{B}$



Programming Demo #1

CUDA Programming w/ Thrust





CUDA Programming w/ Thrust



- Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL):
 - High-level
 - Enhances productivity
 - Allows for interoperability
 - Helps develop high-performance applications

CUDA Programming w/ Thrust



Remember, CUDA programs have a basic flow:
1)The host initializes an array with data.
2)The array is copied from the host to the memory on the CUDA device.
3)The CUDA device operates on the data in the array.
4)The array is copied back to the host.

• This is true for Thrust, too!



Dot Product Example...

- Recall the dot product example from last time:
 - Given vectors a and b each with size N, store the result in scalar c

$$\boldsymbol{c} = \mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \ldots + a_N b_N$$

Purpose of the exercise: use thrust to get it done



• Stage 1: The host initializes the array with data, the code looked like

```
// Allocate host data
float *h_A = (float *) malloc(size);
float *h_B = (float *) malloc(size);
float *h_C = (float *) malloc(size);
float *dotProd_h = (float *)malloc(sizeof(float));
// Initialize the host input vectors
for (int i = 0; i < numElements; ++i) {
    h_A[i] = rand()/(float)RAND_MAX;
    h_B[i] = rand()/(float)RAND_MAX;
}</pre>
```

• Stage 1: Using Thrust, we can change the code to:

```
// Allocate the host vectors
thrust::host_vector<float> h_A;
thrust::host_vector<float> h_B;
thrust::host_vector<float> h_C;
// Initialize the host input vectors
for (int i = 0; i < N; ++i) {
    h_A.push_back(rand()/(float)RAND_MAX);
    h_B.push_back(rand()/(float)RAND_MAX);
    h_C.push_back(0.f);
}</pre>
```

• Stage 2: Data copied from host to device memory; code looked like

```
// Allocate memory for the device data
float *d_A = NULL;
float *d_B = NULL;
float *d_C = NULL;
float *dotProd_d = NULL;
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);
cudaMalloc((void **)&dotProd_d, sizeof(float));
```

// Copy the host input vectors A and B in host memory
// to the device input vectors in device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

• Stage 2: Using thrust, the code can be simplified:

// Allocate the device vectors
thrust::device_vector<float> d_A = h_A;
thrust::device_vector<float> d_B = h_B;
thrust::device_vector<float> d_C = h_C;

• Keep in mind that what happens under the hood is the same copy of data from the host to the device; i.e., it's still an expensive operation

 Stage 3: The CUDA device operates on the data in the array, we originally have the following code:

```
// Launch the Vector Dot Product CUDA Kernel
int threadsPerBlock = numElements;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
vectorDot<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, dotProd_d, numElements);
```



• Stage 3: Using Thrust, we can change the code to:

- d_C.begin(), d_C.end());
- Note that we do not need to specify the execution configuration

- Stage 4: The value is copied back to the host, code looks like:

// Copy the device result vector in device memory to the
// host result vector in host memory.
cudaMemcpy(dotProd_h, dotProd_d, sizeof(float), cudaMemcpyDeviceToHost);

- We can completely remove this step since thrust::reduce(...) copies this for us
- Thrust also cleans up after itself, no need to include free(...) or cudaFree(...)



- To compile this code:
- \$ nvcc dotProductThrust.cu

- To run this code:
- \$ qsub submit_example.sh



Programming Demo #2