

Lecture 25: GPU programming

David Bindel

28 Apr 2010

Logistics

Reminder:

- ▶ Last slot (5/5): *brief* project presentations
 - ▶ Tell me (and your fellow students) what you're up to
 - ▶ Keep to about 5 minutes – slides or board
- ▶ Project reports due by 5/20 at latest
 - ▶ *Don't* make me read a ton of code
 - ▶ *Don't* ask for an extension (pretty please!)
 - ▶ *Do* show speedup plots, timing tables, profile results, models, and anything else that shows you're thinking about performance
 - ▶ *Do* tell me how this work might continue given more time
- ▶ Informal course evaluation questions
 - ▶ What things were particularly hard/easy?
 - ▶ What things did you wish you saw more/less of?
 - ▶ What things did you learn “accidentally”?
 - ▶ What else might make the class better next time?

Some history

- ▶ Late 80s-early 90s: “golden age” for supercomputing
 - ▶ Companies: Thinking Machines, MasPar, Cray
 - ▶ Relatively fast processors (vs memory)
 - ▶ Lots of academic interest and development
 - ▶ *But* got hard to compete with commodity hardware
 - ▶ Scientific computing is not a market driver!
- ▶ 90s-early 2000s: age of the cluster
 - ▶ Beowulf, grid computing, etc.
 - ▶ “Big iron” also uses commodity chips (better interconnect)
- ▶ Past few years
 - ▶ CPU producers move to multicore
 - ▶ High-end graphics becomes commodity HW
 - ▶ Gaming *is* a market driver!
 - ▶ GPU producers realize their many-core designs can apply to general purpose computing

Thread design points

- ▶ Threads on desktop CPUs
 - ▶ Implemented via lightweight processes (for example)
 - ▶ General system scheduler
 - ▶ Thrashing when more active threads than processors
- ▶ An alternative approach
 - ▶ Hardware support for many threads / CPU
 - ▶ Modest example: hyperthreading
 - ▶ More extreme: Cray MTA-2 and XMT
 - ▶ Hide memory latency by thread switching
 - ▶ *Want* many more independent threads than cores
- ▶ GPU programming
 - ▶ Thread creation / context switching are basically free
 - ▶ *Want lots* of threads (thousands for efficiency?!)

General-purpose GPU programming

- ▶ Old GPGPU model: use texture mapping interfaces
 - ▶ People got good performance!
 - ▶ But too clever by half
- ▶ CUDA (Compute Unified Device Architecture)
 - ▶ More natural general-purpose programming model
 - ▶ Initial release in 2007; now in version 3.0
- ▶ OpenCL
 - ▶ Relatively new (late 2009); in Apple's Snow Leopard
 - ▶ Open standard (Khronos group) – includes NVidia, ATI, etc
- ▶ And so on: DirectCompute (MS), Brook+ (Stanford/AMD), Rapidmind (Waterloo (Sh)/Rapidmind/Intel?)

Today: C for CUDA (more available examples)

Compiling CUDA

- ▶ `nvcc` is the driver
 - ▶ Builds on top of `g++` or other compilers
- ▶ `nvcc` driver produces CPU and PTX code
- ▶ PTX (Parallel Thread eXecution)
 - ▶ Virtual machine and ISA
 - ▶ Compiles down to binary for target
- ▶ Can compile in *device emulation mode* for debug
 - ▶ `nvcc -deviceemu`
 - ▶ Can use native debug support
 - ▶ Can access data across host/device boundaries
 - ▶ Can call `printf` from device code

CUDA programming

```
do_something_on_cpu();  
some_kernel<<<nBlk, nTid>>>(args);  
do_something_else_on_cpu();  
cudaThreadSynchronize();
```

- ▶ Highly parallel *kernels* run on device
 - ▶ Vaguely analogous to parallel sections in OpenMP code
- ▶ Rest of the code on host (CPU)
- ▶ C + extensions to program both host code and kernels

Thread blocks

- ▶ Monolithic thread array partitioned into blocks
 - ▶ Blocks have 1D or 2D numeric identifier
 - ▶ Threads within blocks have 1D, 2D, or 3D identifier
 - ▶ Identifiers help figure out what data to work on
- ▶ Blocks cooperate via shared memory, atomic ops, barriers
- ▶ Threads in different blocks cannot cooperate
 - ▶ ... except for implied global barrier from host

Memory access

- ▶ *Registers* are registers; per thread
- ▶ *Shared* memory is small, fast, on-chip; per block
- ▶ *Global* memory is large uncached off-chip space
 - ▶ Also accessible by host

Also runtime support for texture memory and constant memory.

Basic usage

1. Perform any needed allocations
2. Copy data from host to device
3. Invoke kernel
4. Copy results from device to host
5. Clean up allocations

Device memory management

```
h_data = malloc(size);  
... Initialize h_data on host ...  
cudaMalloc((void**) &d_data, size);  
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);  
... invoke kernel ...  
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);  
cudaFree(d_data);  
free(h_data);
```

Notes:

- ▶ Don't dereference h_data on device or d_data on host!
- ▶ Can also copy host-to-host, device-to-device
- ▶ Kernel invocation is asynchronous with CPU; cudaMemcpy is synchronous
(can synchronize kernels with cudaThreadSynchronize)

CUDA function declarations

```
__device__ float device_func();  
__global__ void kernel_func();  
__host__ float host_func();
```

- ▶ `__global__` for kernel (must return void)
- ▶ `__device__` functions called and executed on device
- ▶ `__host__` functions called and executed on host
- ▶ `__device__` and `__host__` can be used together

Restrictions on device functions

- ▶ No taking the address of a `__device__` function
- ▶ No recursion
- ▶ No static variables inside the function
- ▶ No varargs

Kernel invocation

Kernels called with an *execution configuration*:

```
__global__ void kernel_func(...);  
dim3 dimGrid(100, 50); // 5000 thread blocks  
dim3 dimBlock(4, 8, 8); // 256 threads per block  
size_t sharedMemBytes = 64;  
kernel_func<<dimGrid, dimBlock, sharedMemBytes>>(...);
```

- ▶ Can write integers (1D layouts) for first two arguments
- ▶ Third argument is optional (defaults to zero)
- ▶ Optional fourth argument for stream of execution
 - ▶ Used to specify asynchronous execution across kernels
- ▶ Kernel can fail if you request too many resources

Example: Vector addition

```
__global__ void
VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
int threadsPerBlock = 256;
int blocksPerGrid = (N+255) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B,d_C,N);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

Shared memory

Size known at compile time

```
__global__ void kernel(...)  
{  
    __shared__ float x[256];  
    ...  
}
```

```
kernel<<<nb,bs>>>(...);
```

Size known at kernel launch

```
__global__ void kernel(...)  
{  
    extern __shared__ float x[];  
    ...  
}
```

```
kernel<<<nb,bs,bytes>>>(...);
```

Synchronize access with barrier.

Example: Butterfly reduction

```
__global__ void sum_reduce(int* x)
{
    // B is a compile time constant power of 2
    int i = threadIdx.x;
    __shared__ int sum[B];
    sum[i] = x[i]; __syncthreads();
    for (int bit = B/2; bit > 0; bit /= 2) {
        int inbr = (i + bit) % B;
        int t = sum[i] + sum[inbr]; __syncthreads();
        sum[i] = t; __syncthreads();
    }
}

sum_reduce<<1,N>>(d_x);
```

General picture: CUDA extensions

- ▶ Type qualifiers:
 - ▶ global
 - ▶ device
 - ▶ shared
 - ▶ local
 - ▶ constant
- ▶ Keywords (`threadIdx`, `blockIdx`)
- ▶ Intrinsic (`__syncthreads`)
- ▶ Runtime API (memory, symbol, execution management)
- ▶ Function launch

Libraries and languages

The usual array of language tools exist:

- ▶ CUBLAS, CUFFT, CUDA LAPACK bindings (commercial)
- ▶ CUDA-accelerated libraries (e.g. in Trilinos)
- ▶ Bindings to CUDA from Python, Java, etc

Hardware picture (G80)

- ▶ 128 processors execute threads
- ▶ Thread Execution Manager issues threads
- ▶ Parallel data cache / shared memory per processor
- ▶ All have access to device memory
 - ▶ Partitioned into global, constant, texture spaces
 - ▶ Read-only caches to texture and constant spaces

HW thread organization

- ▶ Single Instruction, Multiple Thread
- ▶ A *warp* of threads executes *physically* in parallel (one warp == 32 parallel threads)
- ▶ Blocks are partitioned into warps by consecutive thread ID
- ▶ Best efficiency when all threads in warp do same operation
 - ▶ Conditional branches reduce parallelism — serially execute all paths taken

Memory architecture

- ▶ Memory divided into 16 banks of 32-byte words
- ▶ Each bank services one address per cycle
- ▶ Conflicting accesses are serialized
- ▶ Stride 1 (or odd stride): no bank conflicts

Batch memory access: coalescing

- ▶ *Coalescing* is a coordinated read by half-warp
- ▶ Read contiguous region (64, 128, or 256 bytes)
- ▶ Starting address for region a multiple of region size
- ▶ Thread k in half-warp accesses element k of blocks
- ▶ Not all threads need to participate

The usual picture

- ▶ Performance is potentially quite complicated!
 - ▶ ... and memory is important.
- ▶ Fortunately, there are profiling tools included
- ▶ Unfortunately, I have yet to play with them!

Resources

Beside the basic NVidia documentation, see:

- ▶ http://developer.nvidia.com/object/cuda_training.html
- ▶ <http://courses.ece.illinois.edu/ece498/al/>
- ▶ <http://gpgpu.org/developer>