Introduction to GPU Computing Using CUDA

Spring 2014 Westgid Seminar Series

Scott Northrup
SciNet
www.scinethpc.ca

(Slides http://support.scinet.utoronto.ca/∼northrup/westgrid_CUDA.pdf)

March 12, 2014

Outline

Outline

Heterogeneous Computing

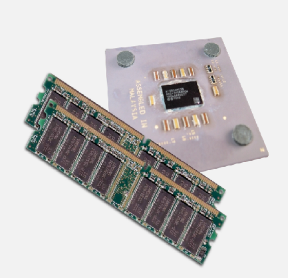
What is it?

- Use different compute device(s) concurrently in the same computation.
- Example: Leverage CPUs for general computing components and use GPU's for data parallel / FLOP intensive components.
- Pros: Faster and cheaper ($/FLOP/Watt) computation
- Cons: More complicated to program

Heterogeneous Computing

Terminology

- GPGPU : General Purpose Graphics Processing Unit
- HOST : CPU and its memory
- DEVICE : Accelerator (GPU) and its memory



Host

Device

Outline

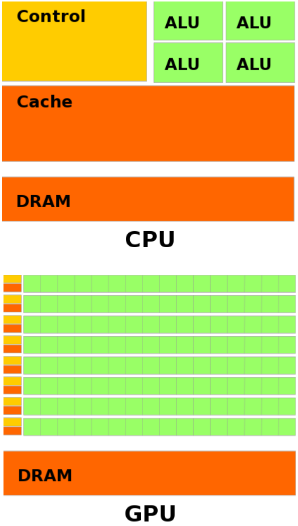
GPU vs. CPUs

CPU

- general purpose
- task parallelism (diverse tasks)
- maximize serial performance
- large cache
- multi-threaded (4-16)
- some SIMD (SSE, AVX)

GPU

- data parallelism (single task)
- maximize throughput
- small cache
- super-threaded (500-2000+)
- almost all SIMD



Speedup

What kind of speedup can I expect?

- $\sim$1 TFLOPs per GPU vs. $\sim$100 GFLOPs multi-core CPU
- 0x - 50x reported

Speedup

What kind of speedup can I expect?

- $\sim$1 TFLOPs per GPU vs. $\sim$100 GFLOPs multi-core CPU
- 0x - 50x reported

Speedup depends on

- problem structure
  - need many identical independent calculations
  - preferably sequential memory access
- single vs. double precision (K20 3.52 TF SP vs 1.17 TF DP)
- level of intimacy with hardware
- time investment

GPU Programming

GPGPU Languages

- OpenGL, DirectX (Graphics only)
- OpenCL (1.0, 1.1, 2.0) Open Standard
- CUDA (NVIDIA proprietary)
- OpenACC
- OpenMP 4.0

Compute Unified Device Architecture

- parallel computing platform and programming model created by NVIDIA
- Language Bindings
  - C/C++ `nvcc` compiler (works with gcc/intel)
  - Fortran (PGI compiler)
  - Others (pyCUDA,jCUDA, etc.)
- CUDA Versions (V1.0 - 6.0)
- Hardware Compute Capability (1.0 - 3.5)
  - Tesla M20*0 (Fermi) has CC 2.0
  - Tesla K20 (Kepler) has CC 3.5

Compute Canada Resources

GPU Systems

- Westgrid: Parallel
  - 60 nodes (3x NVIDIA M2070)
- SharcNet: Monk
  - 54 nodes (2x NVIDIA M2070)
- SciNet: Gravity, ARC
  - 49 nodes (2x NVIDIA M2090)
  - 8 nodes (2x NVIDIA M2070)
  - 1 node (1x NVIDIA K20)
- CalcuQuebec: Guillimin
  - 50 nodes (2x NVIDIA K20)

Outline

CUDA Example: Addition

Device "Kernel" code

```
__global__ void add(float *a, float *b, float *c) {
  *c = *a + *b;
}
```

CUDA Syntax: Qualifiers

- `__global__` indicates a function that runs on the DEVICE called from the HOST
- Used by the compiler, `nvcc`, to separate sort HOST and DEVICE components

CUDA Example: Addition

Host Code: Components

- Allocate Host/Device memory
- Initialize Host Data
- Copy Host Data to Device
- Execute Kernel on Device
- Copy Device Data back to Host
- Output
- Clean-up

CUDA Example: Addition

Host code: Memory Allocation

```
int main(void ) {
 float a, b, c; // host copies of a, b, c
 float *da, *db, *dc; // device copies of a, b, c
 int size = sizeof(float );
 // Allocate space for device copies of a, b, c
 cudaMalloc((void **)&da, size);
 cudaMalloc((void **)&db, size);
 cudaMalloc((void **)&dc, size);
 ...
}
```

CUDA Syntax: Memory Allocation

- `cudaMalloc` allocates memory on DEVICE
- `cudaFree` deallocates memory on DEVICE

CUDA Example: Addition

Host code: Data Movement

```
{
  ...
  // Setup input values
  a = 2.0; b = 7.0;
  // Copy inputs to device
  cudaMemcpy(da, &a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(db, &b, size, cudaMemcpyHostToDevice);
  ...
}
```

CUDA Syntax: Memory Transfers

- `cudaMemcpy` copies memory
  - from DEVICE to HOST
  - from HOST to DEVICE

CUDA Example: Addition

Host code: kernel execution

```
{
  ...
  // Launch add() kernel on GPU
  add<<<1,1>>>(da, db, dc);
  ...
}
```

CUDA Syntax: kernel execution

- `<<<N,M>>>` Triple brackets denote a call from HOST to DEVICE
- Come back to `N` , `M` values later

CUDA Example: Addition

Host code: Get Data, Output, Cleanup

```
{
  ...
  // Copy result back to host
  cudaMemcpy(&c, dc, size, cudaMemcpyDeviceToHost);
  printf('' %2.0f + %2.0f = %2.0f '',a,b,c);
  // Cleanup
  cudaFree(da); cudaFree(db); cudaFree(dc);
  return 0;
}
```

Compile and Run

```
$nvcc -arch=sm_20 hello.cu -o hello
$./hello
$2.0 + 7.0 =9.0
```

CUDA Basics: Review

Device "Kernel" code

- `__global__` function qualifier

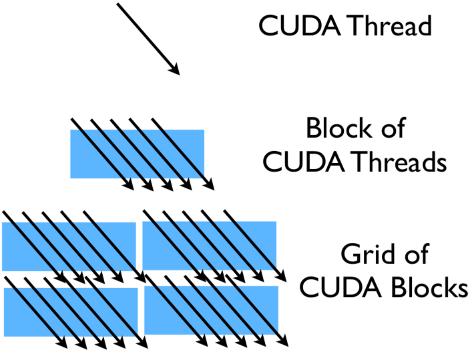
Host Code

- Allocate Host/Device memory `cudaMalloc(...)`
- Initialize Host Data
- Copy Host Data to Device `cudaMemcpy(...)`
- Execute Kernel on Device `fn<<< N, M >>>(...)`
- Copy Device Data to Host `cudaMemcpy(...)`
- Output
- Clean-up `cudaFree(...)`

CUDA Parallelism: Threads, Blocks, Grids

Blocks & Threads

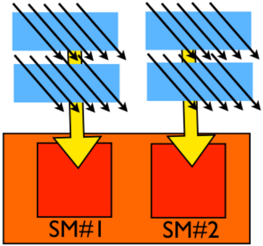
- Threads: execution thread
- Blocks: group of threads
- Grids: set of blocks
- built-in variables to define threads position
  - `threadIdx`
  - `blockIdx`
  - `blockDim`



CUDA Parallelism: Threads, Blocks, Grids

Blocks & Threads

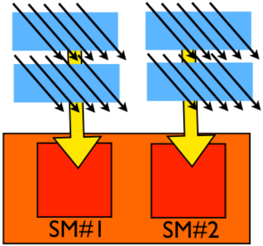
- Threads operate in a SIMD(ish) manner, each execute the same instructions in lockstep
- Blocks are assigned to a GPU, executing one "warp" at a time (usually 32 threads)



CUDA Parallelism: Threads, Blocks, Grids

Blocks & Threads

- Threads operate in a SIMD(ish) manner, each execute the same instructions in lockstep
- Blocks are assigned to a GPU, executing one "warp" at a time (usually 32 threads)



Kernel execution

```
fn<<< blockspergrid, threadsperblock >>>(...)
```

CUDA Example: Vector Addition

Host code: Allocate Memory

```
int main(void ) {
  int N=1024; //size of vector
  float *a, *b, *c; // host copies of a, b, c
  float *da, *db, *dc; // device copies of a, b, c
  int size = N*sizeof(float );
  // Allocate space for host copies of a, b, c
  a = (float *)malloc (size);
  b = (float *)malloc (size);
  c = (float *)malloc (size);
  // Allocate space for device copies of a, b, c
  cudaMalloc((void **)&da, size);
  cudaMalloc((void **)&db, size);
  cudaMalloc((void **)&dc, size);
  ...
}
```

CUDA Example: Vector Addition

Host code: Initialize and Copy

```
{
 ...
 // Setup input values
 for (int i=0;i<N;i++){
  a[i] = (float )i;
  b[i] = 2.0*(float )i;
 }
 // Copy inputs to device
 cudaMemcpy(da, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(db, b, size, cudaMemcpyHostToDevice);

 // Launch add() kernel on GPU
 add<<<1,N>>>(da, db, dc);
 ...
}
```

CUDA Example: Vector Addition

Host code: Get Data, Output, Cleanup

```
{
  ...
  // Copy result back to host
  cudaMemcpy(c, dc, size, cudaMemcpyDeviceToHost);
  printf(``Hello World!, I can add on a GPU'');
  for (int i=0;i<N;i++){
   printf("%d %2.0f + %2.0f = %2.0f",i,a[i],b[i],c[i]);
  }
  // Cleanup
  free(a); free(b); free(c);
  cudaFree(da); cudaFree(db); cudaFree(dc);
  return 0;
}
```

SciNet compute • calcul CANADA

CUDA Threads

Kernel using just threads

```
__global__ void add(float *a, float *b, float *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Host code: kernel execution

```
...
// Launch add() kernel on GPU
// with 1 block and N threads
add<<<1,N>>>(da, db, dc);
...
```

SciNet compute • calcul CANADA

CUDA Blocks

Kernel using just blocks

```
__global__ void add(float *a, float *b, float *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

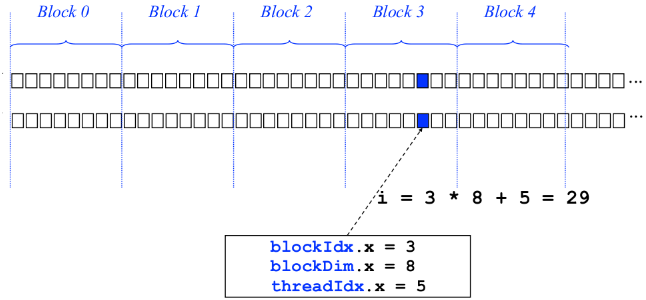
Host code: kernel execution

```
...
// Launch add() kernel on GPU
// with N blocks, 1 thread each
add<<<N,1>>>(da, db, dc);
...
```

CUDA Blocks & Threads

Indexing with Blocks and Threads

- Use built-in variables to define unique position
  - `threadIdx` : thread ID (within a block)
  - `blockIdx` : block ID
  - `blockDim` : threads per block



CUDA Blocks & Threads

kernel using blocks

```
__global__ void add(float *a, float *b, float *c, int n) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if(index < n);
  c[index] = a[index] + b[index];
}
```

Host code: kernel execution

```
...
int TB=128; //threads per block
// Launch add() kernel on GPU
add<<<N/TB,TB>>>(da, db, dc,N);
...
```

Outline

More CUDA Syntax

Qualifiers

- Functions
  - `__global__` : Device kernels called from host
  - `__host__` : Host only (default)
  - `__device__` : Device only called from device
- Data
  - `__shared__` : Memory shared within a block
  - `__constant__` : Special memory for constants (cached)
- Control
  - `__syncthreads()`: thread barrier within a block

More CUDA Syntax

Qualifiers

- Functions
  - `__global__` : Device kernels called from host
  - `__host__` : Host only (default)
  - `__device__` : Device only called from device
- Data
  - `__shared__` : Memory shared within a block
  - `__constant__` : Special memory for constants (cached)
- Control
  - `__syncthreads()`: thread barrier within a block

More details

- Grids and Blocks can be 1D, 2D, or 3D (type `dim3`)
- Error Handling: `cudaError_t cudaGetLastError(void)`
- Device Management: `cudaGetDeviceProperties(...)`

CUDA Kernels

Kernel Limitations

- No recursion in `__host__`, allowed in `__device__` for CC > 2.0
- No variable argument lists
- No dynamic memory allocation
- No function pointers
- No static variables inside kernels

CUDA Kernels

Kernel Limitations

- No recursion in `__host__`, allowed in `__device__` for CC > 2.0
- No variable argument lists
- No dynamic memory allocation
- No function pointers
- No static variables inside kernels

Performance Tips

- Exploit parallelism
- Avoid branches in device code
- Avoid memory transfers between Device and Host
- GPU memory
  - high bandwidth/high latency
  - can hide latency with lots of threads
  - access patterns matter (coalesced)

CUDA Libraries & Applications

Libraries

- CUBLAS
- CULA
- CUSPARSE
- CUFFT
- https://developer.nvidia.com/gpu-accelerated-libraries

CUDA Libraries & Applications

Libraries

- CUBLAS
- CULA
- CUSPARSE
- CUFFT
- https://developer.nvidia.com/gpu-accelerated-libraries

Applications

- GROMACS, NAMD, LAMMPS, AMBER, CHARMM
- WRF, GEOS-5
- Fluent 15.0, ANSYS, Abaqus
- Matlab, Mathematica
- http://www.nvidia.com/object/gpu-applications.html

Outline

Intro to CUDA

Summary

- Hetergeneous Computing
- CUDA Basics
  - `__global__` , `cudaMemcpy(...)`, `fn<<<blocks,threads_per_block>>>(...)`
  - blocks, threads
  - indexing (`threadIdx.x`, `blockIdx.x` , `blockDim.x`)
  - Limitations
  - Performance
- CUDA Libraries and Applications
- https://developer.nvidia.com/cuda