Lecture 34: GPGPU Programming with CUDA (contd)

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for Today’s Lecture

- Handout for Lecture 33
- Slides from CS6963: Parallel Programming for GPUs, Mary Hall, Department of Computer Science, University of Utah
  - http://www.cs.utah.edu/~mhall/cs6963s10
- Sanjay Chatterjee, Graduate TA, COMP 322
Announcements

• Homework 7 due by 5pm on Friday, April 22\textsuperscript{nd}
  – Send email to comp322-staff if you’re running into issues with accessing SUG@R nodes, or anything else
Process Flow of a CUDA Kernel Call (Figure 2)

- Data parallel programming architecture from NVIDIA
  - Execute programmer-defined kernels on extremely parallel GPUs
  - CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - cudaMalloc, cudaMemcpy, cudaFree, etc.

Figure source: Y. Yan et. al. “JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA.” Euro-Par 2009.
Organization of a CUDA grid (Figure 4)
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Block ID = index of outer forall
- Thread ID = index of inner forall
CUDA Global Memory

Global memory

- Main means of communicating R/W Data between host and device
- Contents visible to all threads
- Long latency access
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device Global Memory
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of the allocated object

- **cudaFree()**
  - Frees object from device Global Memory
CUDA Host-Device Data Transfer

- **cudaMemcpy()**
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer
CUDA Host-Device Data Transfer

- cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)

- copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- The memory areas may not overlap

- Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.
Host Code in C for Matrix Multiplication

1. void MatrixMultiplication(float* M, float* N, float* P, int Width) {
    2.     int size = Width*Width*sizeof(float); // matrix size
    3.     float* Md, Nd, Pd; // pointers to device arrays
    4.     cudaMalloc((void**)&Md, size); // allocate Md on device
    5.     cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
    6.     cudaMalloc((void**)&Nd, size); // allocate Nd on device
    7.     cudaMemcpy(Nd, M, size, cudaMemcpyHostToDevice); // copy N to Nd
    8.     cudaMalloc((void**)&Pd, size); // allocate Pd on device
    9.     dim3 dimBlock(Width,Width); dim3 dimGrid(1,1);
   10.    // launch kernel (equivalent to “async at(GPU), forall, forall”
   11.    MatrixMulKernel<<<dimGrid,dimBlock>>>(Md, Nd, Pd, Width);
   12.    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
   13.    // Free device matrices
   14.    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
   15. }

COMP 322, Spring 2011 (V. Sarkar)
Matrix multiplication kernel code in CUDA (Figure 6)

```c
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
    float Mdelement = Md[ty * Width + k];
    float Ndelement = Nd[k * Width + tx];
    Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```
CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function (must return void)
- __device__ and __host__ can be used together
- __device__ functions have a number of restrictions
  - No indirect function calls
  - No recursion (restriction being removed in latest release)
  - No static variable declarations inside the function
CUDA Compiler’s Role: Partition Code and Compile for Device

mycode.cu

int main_data;
__shared__ int sdata;

Main() { }
__host__ hfunc () {
    int hdata;
    <<<gfunc(g,b,m)>>>(
}

__global__ gfunc() {
    int gdata;
}

__device__ dfunc() {
    int ddata;
}

Compiled by native compiler: gcc, icc, cc

int main_data;

Main() { }
__host__ hfunc () {
    int hdata;
    <<<gfunc(g,b,m)>>>(
}

__global__ gfunc() {
    int gdata;
}

__device__ dfunc() {
    int ddata;
}

Compiled by nvcc compiler

int main_data;
__shared__ sdata;

Main() { }
__host__ hfunc () {
    int hdata;
    <<<gfunc(g,b,m)>>>(
}

__global__ gfunc() {
    int gdata;
}

__device__ dfunc() {
    int ddata;
}
CUDA Storage Classes

- **Device code can:**
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory

- **Host code can**
  - Transfer data to/from per-grid global and constant memories
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    __global__ void KernelFunc(float* ptr)
  - Obtained as the address of a global variable: float* ptr = &GlobalVar;
Usage Patterns for CUDA Storage Classes

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication
Constant Memory Example

- Signal recognition:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute $M_1V, M_2V, \ldots, M_nV$

```c
__constant__ float d_signalVector[M];
__device__ float R[N][M];

__host__ void outerApplySignal () {
    float *h_inputSignal;
    dim3 dimGrid(N);
    dim3 dimBlock(M);
    cudaMemcpyToSymbol (d_signalVector, h_inputSignal, M*sizeof(float));
    // input matrix is in d_mat
    ApplySignal<<<dimGrid,dimBlock>>>(d_mat, M);
}

__global__ void ApplySignal (float * d_mat, int M) {
    float result = 0.0; /* register */
    for (j=0; j<M; j++)
        result += d_mat[blockIdx.x][threadIdx.x][j] * d_signalVector[j];
    R[blockIdx.x][threadIdx.x] = result;
}
```
Use of Constant Cache for Constant Memory

- Example from previous slide
  - All threads in a block accessing same element of signal vector
  - Brought into cache for first access, then latency equivalent to a register access

```
P_0
Reg

P_1
Reg

P_{M-1}
Reg

Constant Cache

Instruction Unit

LD signalVector[j]
```
Impact of Single Control Unit for a Block of Threads executing on an SM

Control flow example
if (threadIdx >= 2) {
    out[threadIdx] += 100;
} else {
    out[threadIdx] += 10;
}

SIMD = Single Instruction Multiple Data
SIMD Execution of Control Flow

Control flow example
if (threadIdx.x >= 2) {
    out[threadIdx.x] += 100;
} else {
    out[threadIdx.x] += 10;
}

/* Condition code cc = true branch set by predicate execution */
(CC) LD R5, &(out +threadIdx.x)
(CC) ADD R5, R5, 100
(CC) ST R5, &(out +threadIdx.x)
SIMD Execution of Control Flow

Control flow example

if (threadIdx >= 2) {
    out[threadIdx] += 100;
}
else {
    out[threadIdx] += 10;
}

/* possibly predicated using CC */
(not CC) LD R5, &out[threadIdx]
(not CC) ADD R5, R5, 10
(not CC) ST R5, &out[threadIdx]
Divergence

• Divergent paths
  — What happens if different threads within a block take different control flow paths?
  — N divergent paths
    - An N-way divergent block is serially issued over the N different paths
    - Performance decreases by about a factor of N
    - GPU is better suited for blocks of threads with low intra-block divergence
    - Multicore CPU is better equipped to handle divergence than CPU

• Implementation note
  — Current GPUs subdivide a block of threads into “warps” of a fixed size (e.g., 32 or 64)
  — Divergence can be tolerated among threads in different warps, but not among threads in the same warp
  — If you avoid divergence within a block, you will also guarantee the absence of divergence within a warp
Summary: GPU Pros and Cons

Pros

• Efficient support for large numbers of threads
• Potential for 10x – 100x performance improvement relative to multicore CPUs for certain classes of applications
• CUDA provides a simple software abstraction of GPUs

Cons

• SIMD execution punishes control-flow divergence within a block
• Availability of cache is limited and restricted
• All pointer data structures must be stored in global memory (even if access is thread-local)
• Large number of local scalar variables can degrade performance due to “register spills”
• Very limited synchronization available across threads (only __syncthreads() barrier for threads in the same block)

Future trend

• CPUs and GPUs will be integrated in a single chip, and many of the current GPU restrictions will go away