Lecture 19

GPU Applications
Performance Programming
Heterogeneous and Hybrid Processing
Announcements

• Quiz on Thursday
# MPI Behavior

<table>
<thead>
<tr>
<th></th>
<th>Process P0</th>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IRecv(from:*, r1)</td>
<td>Compute</td>
<td>Compute</td>
</tr>
<tr>
<td>2</td>
<td>IRecv(from:2, r2)</td>
<td>Send(to:0, s1)</td>
<td>Send(to:0, s2)</td>
</tr>
<tr>
<td>3</td>
<td>Wait for all IRecvs to complete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Today’s lecture

• Applications on the GPU
• Performance Programming
• Multi-tier (hybrid models)
NVIDIA GeForce GTX 280
Hierarchical Thread Organization

- Grid \supseteq Block \supseteq Thread
- Thread Blocks
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
  - Blocks within a grid are virtualized, too
  - May configure number of threads in a block and the number of blocks
- Threads assigned to SM in units of blocks, up to 8 for each SM
- Compiler re-arranges loads to hide latencies

KernelA<<<2,3>,<3,5>>>

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
Memory Hierarchy

<table>
<thead>
<tr>
<th>Name</th>
<th>Latency (cycles)</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Local</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Constant</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
Thread scheduling

• Blocks are divided into *warps* of 32 threads (schedulable unit)
  ‣ SM schedules ready warps, scoreboard, prioritized, zero overhead
  ‣ All threads in a warp execute the same instruction: all branches followed: serialization, instructions may be disabled, divergence

• An SM may be assigned up to 8 thread blocks over 32 warps, max 1024 threads
• A minimum number of warps needed to hide memory latency
• A kernel with different block sizes
  • 8x8: 64 threads/block, 12 blocks, only 512 threads/SM
  • 16x16, 256 threads, 3 blocks, may not be able to hide latency
• Consider another application that uses 32 registers (floats) per thread
• Each SM gets 512 registers, 64 per core, how many threads/SM?
Bank Conflicts

- To improve bandwidth, we organize memory into *banks*
- Each bank responds once per cycle
- We have a *bank conflict* if there is more than 1 access to the same bank in a cycle (but no conflict with Broadcast)
- Accesses are *coalesced* if accesses from a half warp are conflict-free

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.f;
```

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC
Conflict free access

• Conflict free if $s$ has no common factors with the # of banks

```c
__shared__ float A[256];
float foo = A[base + s * threadIdx.x];
```

• What can we say about $s$?
Identifying conflicts

• Consider
  __shared__ float shared[256];

• Traditional wisdom for exploiting cache locality can result in bank conflicts
• If a thread loads 2 consecutive elements..
  
  ```
  int tid = threadIdx.x;
  shared[2*tid] = global[2*tid];
  shared[2*tid+1] = global[2*tid+1];
  ```

• To avoid conflicts
  
  ```
  shared[tid] = global[tid];
  shared[tid + blockDim.x] = global[tid + blockDim.x];
  ```
Summation: bank conflicts and thread divergence

for (i=0; i < N; i++) sum = sum + x[i]

__shared__ int x[ ];
unsigned int tid = threadIdx.x;
unsigned int s;

for ( s = 1; s< blockDim.x ; s*=2){
    __syncthreads();
    if ( tid % (2*s) == 0 )
        x[tid] = x[tid] + x[tid+s];
}

reduceSum <<<N/512,512>>> (x,N)
Conflicts

Array elements

0 1 2 3 4 5 6 7 8 9 10 11
Optimized code - reduced effects of divergence

```cpp
__shared__ int x[ ];
unsigned int tid = threadIdx.x;
unsigned int s;

for ( s = blockDim.x >> 1; s > 0 ; s >>= 1 ){
    __syncthreads();
    if ( tid < s )
        x[tid] = x[tid] + x[tid+s];
}
```

All threads in a warp execute the same instruction

```
reduceSum <<<N/512,512>>> (x,N)
```
Avoiding Divergence and Bank Conflicts

Threads: 0, 1, ...

for ( s = blockDim.x >> 1; s > 0 ; s >>= 1 ){
    __syncthreads();
    if ( tid < s )
        x[tid] = x[tid] + x[tid+s];
}

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
Aliev-Panfilov Stencil Method
Using shared memory in stencil methods

Didem Unat
Kernel – Part I

__shared__ float block[DIM_Y + 2 ][DIM_X + 2 ];
int idx = threadIdx.x, idy = threadIdx.y ; //local indices
//global indices
int x = blockIdx.x * (DIM_X) + idx;
int y = blockIdx.y * (DIM_Y) + idy;
idy++; idx++;
unsigned int index =  y * N +  x  ;

//interior points
float center = E_prev[index] ;
block[idy][idx] = center;

__syncthreads();
Copying the ghost cells

```c
if (idy == 1 && y > 0 )
    block[0][idx]= E_prev[index - N];
else if(idy == DIM_Y && y < N-1)
    block[DIM_Y+1][idx] = E_prev[index + N];
if ( idx==1 && x > 0 )
    block[idy][0] = E_prev[index - 1];
else if( idx== DIM_X && x < N-1 )
    block[idy][DIM_X +1] = E_prev[index + 1];
__syncthreads();
```
The Stencil computation and the ODE

float = R[index];

float \( e = \) center + \( \alpha \) * (block[idy][idx-1] + block[idy][idx+1] + block[idy-1][idx] + block[idy+1][idx] - 4*center);

\( e = e - dt*(kk * e * ( e - a) * ( e - 1 ) + e * r); \)

\( E[index] = e; \)

\( R[index] = r + dt*(\epsilon+ M1 * r / ( e + M2 ) ) * ( -r - kk * e * (e - b - 1)); \)
## Performance

=====Timings for Explicit Aliev-Panfilov Equation =====

N=256, T=1.000000e+03, dt=4.055496e-02

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time</th>
<th>Gflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>1.350 sec</td>
<td>33.527</td>
</tr>
<tr>
<td>Host</td>
<td>28.047 sec</td>
<td>1.613</td>
</tr>
</tbody>
</table>

Speedup = 20.8
Multi-tier (hybrid) Computing
Hiding communication
Hierarchical organization

- Each node is a multiprocessor, usually an SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All multicomputer systems today
- Hybrid programming or message passing
Hybrid MPI + thread programming

- Take advantage of faster shared memory on-node
- Higher complexity than “flat MPI”
- KeLP2: www.cse.ucsd.edu/groups/hpcl/scg/Research/MT.html
Multi-tier model

- Programs have 3 levels of control:
  - **Collective** level operations performed on all nodes
  - **Node** level operations performed on one node
  - Processor level operations performed on a single CPU
Heterogeneous processing

• Two types of processors: general purpose + accelerator
• Accelerator can perform certain tasks more quickly subject to various overhead costs
• Accelerator amplifies relative cost of communication
Communication tolerant algorithms

• 2 Coping strategies
• Reformulate the algorithm, trade computation for communication [SC03]
• Overlap communication with computation [ISCOPE97, SC98, HPDC06….]
A Crosscutting issue: Hiding latency

- Little’s law [1961]
  - The number of threads must equal the parallelism times the latency
    \[ T = p \times \lambda \]
  - \( p \) and \( \lambda \) are increasing with time
- Difficult to implement
  - Split phase algorithms
  - Partitioning and scheduling
- The state-of-the-art enables but doesn’t support the activity
- Distracts from the focus on the domain science
- Implementation policies entangled with correctness issues
  - Non-robust performance
  - High development costs
Motivating application

- Stencil method
- Building block: iterative solver using Jacobi’s method (7-point stencil)

for (i,j,k) in 1:N x 1:N x 1:N
  \[ u[i][j][k] = \frac{(u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k] + u[i][j+1][k] + u[i][j][k+1] + u[i][j][k-1])}{6}; \]
Domain Decomposition Strategy

• Divide problem into subdomains
• Solve each subdomain locally
• Stitch the solutions together
Classic message passing implementation

• Image smoothing
• Decompose domain into sub-regions, one per process
  ‣ Transmit halo regions between processes
  ‣ Compute inner region after communication completes
• Loop carried dependences impose a strict ordering on communication and computation
Latency tolerant variant

- Only a subset of the domain exhibits loop carried dependences with respect to the halo region
- Subdivide the domain to remove some of the dependences
- We may now sweep the inner region in parallel with communication
- Sweep the annulus after communication finishes
- Multi-tier flow control: N nodes, p processes per node
Overlapped code (KeLP2)

```c
Relax(Distributed_Data X, Mover Communication) {
  Communication.start();
  for each subdomain x in X
    Update x
  }
  Communication.wait();
  // Repeat over the annulus ...
}
```

- Implemented with KeLP2 [Fink 98, SC98]
  - Run time system provides message proxy to realize overlap
  - Provides hierarchical control flow
A few implementation details

• Some versions of MPI can realize overlap with MPI_IRecv and MPI_Isend
• If not, then we can use multithreading to handle the overlap
• We let one or more processors (proxy thread(s)) handle communication
Using a proxy
A performance model of overlap

- Assumptions
  
  \( p = \) number of processors per node

  running time = 1.0

  \( f < 1 = \) communication time

  (i.e. not overlapped)
Performance

• When we displace computation to make way for the proxy, computation time increases
• Wait on communication drops to zero, ideally
• When \( f < \frac{p}{2p-1} \): improvement is \( (1-f) \cdot \frac{p}{p-1} \)
• Communication bound: improvement is \( \frac{1}{1-f} \)

\[
T = 1.0 \\
T = (1-f) \cdot \frac{p}{p-1}
\]
NPACI Blue Horizon

• Multiple SMP nodes
  ‣ 144 8-way Power3+ “high” nodes
  ‣ 375 MHz CPU
  ‣ 4 GB memory per node
  ‣ 64 KB L1$, 4MB L2$ per processor
  ‣ Caches have a 128 Byte line size

• Differential MPI communication rates
  (peak Ring)
  ‣ 400 MB/sec off-node
  ‣ 500 MB/sec on node
Performance improves with overlap

- Computation time increases with fewer CPUs
- Loss is mitigated by memory system saturation at high levels of parallelism

Discuss the OPT Variant in detail

[Baden and Fink 98; Fink 98]