Basic GPU Performance

CS378 – Spring 2015

Sreepathi Pai
Outline

- System Performance
- GPU Occupancy
- Data Layout and Work Distribution
- Static Scheduling of Work
System Performance

• GPU + CPU forms a heterogeneous system
  – “A system where programmer must choose where to perform a computation” (definition-in-progress)

• Parallel execution is possible
  – CPU and GPU can be working on work independently in parallel
  – In fact, GPU allows data transfers in parallel to GPU execution

• Consider distributing work so that all execution units (CPU and GPU) are fully occupied

• Not easy to do manually, but no automatic solution widely accepted yet
Measurement Pitfalls

Keep in mind:

1. A GPU program is a \textit{parallel} CPU program
   - i.e. GPU code sometimes runs on a separate thread
2. A CPU + GPU system is a distributed system
   - i.e. clocks are unsynchronized
   - especially across GPU cores
3. Use \textit{timelines} not \textit{intervals} to reason about performance
   - timelines capture overlap
   - timelines illustrate critical path
   - NVIDIA Profiler provides timelines
struct stopwatch va;

clock_start (&va );
vector_add_1 <<<14*8, 384>>>(ga , gb , gc , N );
clockstop (&va );

printf (TIMEFMT "s \n", va.elapsed.tv_sec , va.elapsed.tv_nsec ) ;

• Output is approx. 40μs on my machine

• NVIDIA Compute Profiler:
gputime=[ 14078.336 ] (μs)
Vector-Addition

![Graph showing performance comparison between CPU and GPU for vector addition. The x-axis represents vector size, the y-axis represents time in seconds. The graph shows that CPU performance improves at a slower rate compared to GPU performance.]
Vector Addition + Transfer Time

[Graph showing time vs vector size for CPU, GPU, and GPU+Memcpy]
GPU Occupancy

![Diagram of Warp Scheduler with Instruction Dispatch Units showing different warp instructions and time progression.](image-url)
GPU Occupancy – contd.

- GPUs divide resources among threads to enable hardware multithreading
- The number of *concurrent* threads is determined by the resource that is exhausted first
- Occupancy is the ratio of running concurrent threads to the maximum number of SM threads
- Residency is the number of thread blocks that can run concurrently on the SM
- NVIDIA provides an occupancy calculator that calculates this number for different GPUs

<table>
<thead>
<tr>
<th>Resource</th>
<th>Available</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>2048</td>
<td>1024/block</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>48K (max)</td>
<td>48K/block</td>
</tr>
<tr>
<td>Registers</td>
<td>65536</td>
<td>255/thread</td>
</tr>
<tr>
<td>Thread Blocks</td>
<td>16</td>
<td>16/SM</td>
</tr>
</tbody>
</table>

*Resources are per SM on NVIDIA Kepler*

```
kernel<<<2048, 32>>>(())  
threads/block: 32
registers: 100/thread -> 3200/block
shared mem: 1K/block
residency: 16, exceeds maximum thread blocks
occupancy: 16*32/2048 = 25%
```

```
kernel<<<2048, 32>>>(())  
threads/block: 32
registers: 160/thread -> 5120/block
shared mem: 1K/block
residency: 12, exceeds maximum registers
occupancy: 12*32/2048 = 18%
```
Should occupancy be maximized?

- NVIDIA Manual – roughly, yes
- But:

<table>
<thead>
<tr>
<th>Multiplication of two large matrices, single precision (SGEMM):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CUBLAS 1.1</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Threads per block</td>
</tr>
<tr>
<td>Occupancy (G80)</td>
</tr>
<tr>
<td>Performance (G80)</td>
</tr>
<tr>
<td><strong>8x smaller thread blocks</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch of 1024-point complex-to-complex FFTs, single precision:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CUFFT 2.2</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Threads per block</td>
</tr>
<tr>
<td>Occupancy (G80)</td>
</tr>
<tr>
<td>Performance (G80)</td>
</tr>
<tr>
<td><strong>4x smaller thread blocks</strong></td>
</tr>
</tbody>
</table>

Volkov, V., “Better Performance at Lower Occupancy”, GTC 2010
Volkov's Summary

- Do more parallel work per thread to hide latency with fewer threads (i.e. increase ILP)
- Use more registers per thread to access slower shared memory less
- Both may be accomplished by computing multiple outputs per thread

[Note that Volkov underutilizes threads, but maxes out registers!]
Data Layout

```c
struct pt {
    int x;
    int y;
};

__global__
void aos_kernel(int n_pts, struct pt *p) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int nthreads = blockDim.x * gridDim.x;
    for(int i = tid; i < n_pts; i += nthreads) {
        p[i].x = i;
        p[i].y = i * 10;
    }
}

In main():
struct pt *p;
cudaMalloc(&p, ...)
```

Array of Structure (AoS)

```c
struct pt {
    int *x;
    int *y;
};

__global__
void soa_kernel(int n_pts, struct pt p) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int nthreads = blockDim.x * gridDim.x;
    for(int i = tid; i < n_pts; i += nthreads) {
        p.x[i] = i;
        p.y[i] = i * 10;
    }
}

In main():
struct pt p;
cudaMalloc(&p.x, ...)
cudaMalloc(&p.y, ...)
```

Structure of Arrays (SoA)

Which, if any, is faster?
SoA vs AoS Results
Why?
AoS vs SoA memory layout

\[
p[i].x = i;
\]

\[
p[i].y = i \times 10;
\]

\[
p.x[i] = i;
\]

\[
p.y[i] = i \times 10;
\]
Assigning Work to Threads

\[
\text{start} = \text{tid} \times \text{blksize}; \\
\text{end} = \text{start} + \text{blksize}; \\
\text{for}(i = \text{start}; i < N \land i < \text{end}; i++) \\
\quad a[i] = b[i] + c[i]
\]

\[
\text{start} = \text{tid}; \\
\text{for}(i = \text{start}; i < N; i+=n\text{threads}) \\
\quad a[i] = b[i] + c[i]
\]

Blocked

Interleaved/Striped

Which, if any, is faster?
Blocking v/s Striped

![Graph showing blocking vs striped over vector size](image)
Exploiting Spatial Locality (1)  
Texture Cache

- Textures are 2-D images that are “wrapped” around 3-D models
- Exhibit 2-D locality, so textures have a separate cache
- GPU contains a texture fetch unit that non-graphics programs can also use
  - Step 1: map arrays to textures
  - Step 2: replace array reads by tex1Dfetch(), tex2Dfetch()
- Catch: Only read-only data can be cached
  - you can write to the array, but it may not become visible through the texture in the same kernel call
- Easiest way to use textures:
  - const __restrict__ *
Exploiting Spatial Locality (2)

Shared Memory

- “Shared Memory” is on-chip software-managed cache, also known as a scratchpad
- 48K maximum size
- Partitioned among thread blocks
- `__shared__` qualifier places items in shared memory
- Can be used for communicating between threads of the same thread block

```c
__shared__ int x;
if(threadIdx.x == 0)
    x = 1;
__syncthreads(); //required!
printf("%d\n", x);
```
Using Shared Memory (SGEMM)

```c
__shared__ float c_sub[BLOCKSIZE][BLOCKSIZE];

// calculate c_sub
__syncthreads();

// write out c_sub to memory
```
Constant Data

- 64KB of “constant” data
  - not written by kernel

- Suitable for read-only, “broadcast” data

- All threads in a warp read the same constant data item at the same time
  - what type of locality is this?

- Uses: Filter coefficients
Summary of data access performance

- Layout data structures in memory to maximize bandwidth utilization
- Assign work to threads to maximize bandwidth utilization
- Rethink caching strategies
  - identify readonly data
  - identify blocks that you can load into shared memory
  - identify tables of constants
Distributing Regular Work
Scalar Product

• Problem: Given \( n \) pairs of vectors, all \( w \) elements wide, compute the scalar products of all the pairs
  - Multiplications: \( n \times w \)
  - Additions: \( n \times w \)

• How shall we distribute work?
Scheme 1: Split Source Vectors

- Split each vector, and distribute the splits to individual thread blocks
Scheme 2: Split Destination Vector

- Each thread block calculates one scalar product
If only one scheme is used ...
Solution

- Enough work to saturate GPU
- Just not distributed evenly
- Make two versions – whole-GPU and per-thread-block
- Choose between two versions at runtime depending on input size
- See MonteCarlo in the CUDA SDK (4.2) for an example
Conclusion

- Focus on full system performance
- Use GPU resources judiciously
  - don't focus on only maximizing occupancy
- Layout data in memory well
  - SoA usually performs better
  - Take advantage of read-only, blocked, and constant characteristics
- Distribute computation well
  - take memory accesses into account
  - be aware of the pitfalls of static scheduling for different input sizes