CS377P Programming for Performance
Basic GPU Programming

Sreepathi Pai
April 24, 2015

UTCS
Outline

Introduction to CUDA

Basic Performance

Memory Performance
Outline

Introduction to CUDA

Basic Performance

Memory Performance
Background

- Discrete GPUs
- CUDA
• CUDA is a C++ dialect
  • extra keywords
  • extra semantics

• CUDA code consists of:
  • *device* code (executes on the GPU)
  • *host* code (executes on the CPU)
  • execution always starts on the CPU

• CUDA compiler is `nvcc`
void vector_add(int *a, int *b, int *c, int N) {
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(void) {
    ...
    vector_add(a, b, c, N);
}
Kernels: __global__ keyword

```c
__global__
void vector_add(int *a, int *b, int *c, int N) {
    ...
}

int main(void) {
    ...
    vector_add<<<...>>>(a, b, c, N);
}
```

- The __global__ keyword indicates a GPU kernel
- GPU kernels must be called with a *configuration*
GPU kernels are SPMD kernels
  - All threads execute the same code
Number of threads to execute is specified at launch time
  - As a grid of $B$ thread blocks of $T$ threads each
  - Total threads: $B \times T$
Reason: Only threads within the same thread block can communicate with each other (cheaply)
  - Other reasons too, but this is the only algorithm-specific reason
Determining the Configuration: Work Size

- Determine a thread block size: say, 256 threads
- Divide work by thread block size
  - Round up
  - \( \lceil N/256 \rceil \)
- Configuration can be changed every call

```c
int threads = 256;
int Nup = (N + threads - 1) / threads;
int blocks = Nup / threads;

vector_add<<<blocks, threads>>>(...)
```
Distributing work in the kernel

```c
__global__
vector_add(int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if(tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}
```

- Maximum $2^{32}$ threads supported
- `gridDim`, `blockDim`, `blockIdx` and `threadIdx` are CUDA-provided variables
__global__
vector_add(int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if(tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}

int main(void) {
    int threads = 256;
    int Nup = (N + threads - 1) / threads;
    int blocks = Nup / threads;

    ...

    vector_add<<<blocks, threads>>>(a, b, c, N);
}
CUDA vector_add: GPU memory

- GPU can’t read CPU memory directly by default
- Arrays a, b need to be copied to GPU memory
- The result c needs to be copied back to CPU memory
- Two CUDA functions:
  - cudaMemcpy allocates GPU memory
  - cudaMemcpy copies memory between CPU and GPU
CUDA vector_add with GPU memory

```c
int *g_a, *g_b, *g_c;

cudaMalloc(&g_a, sizeof(a[0]) * N);
cudaMalloc(&g_b, sizeof(b[0]) * N);
cudaMalloc(&g_c, sizeof(c[0]) * N);

cudaMemcpy(g_a, a, sizeof(a[0]) * N, cudaMemcpyHostToDevice);
cudaMemcpy(g_b, b, sizeof(b[0]) * N, cudaMemcpyHostToDevice);

vector_add<<<...>>>(g_a, g_b, g_c, N);

cudaMemcpy(c, g_c, sizeof(c[0]) * N, cudaMemcpyDeviceToHost);
```
CUDA `vector_add`: complete?

```c
__global__
vector_add(int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if(tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}

int main(void) {
    int threads = 256;
    int Nup = (N + threads - 1) / threads;
    int blocks = Nup / threads;

    cudaMalloc(&g_a, sizeof(a[0]) * N);
    cudaMalloc(&g_b, sizeof(b[0]) * N);
    cudaMalloc(&g_c, sizeof(c[0]) * N);

    cudaMemcpy(g_a, a, sizeof(a[0]) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(g_b, b, sizeof(b[0]) * N, cudaMemcpyHostToDevice);
    vector_add<<<blocks, threads>>>(g_a, g_b, g_c, N);
    cudaMemcpy(c, g_c, sizeof(c[0]) * N, cudaMemcpyDeviceToHost);
}
```
Outline

Introduction to CUDA

Basic Performance

Memory Performance
Heterogeneous Systems

• GPU + CPU form a heterogeneous system
  • “A system with non-trivial choices of where to perform a computation”
• Parallel execution is possible
  • CPU and GPU can 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 utilized
• Not easy to do manually, but no automatic solution widely accepted yet
Measurement Pitfalls

Keep in mind:

- A GPU program is a parallel CPU program
  - i.e. GPU code sometimes runs on a separate thread
- A CPU + GPU system is a distributed system
  - i.e. clocks are unsynchronized
  - especially across GPU cores
- Use timelines not 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);
clock_stop (&va);

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

• Output is approx. 40\mu s on my machine
• NVIDIA Compute Profiler:
  • gputime=[ 14078.336 ] (\mu s)
Vector Addition Performance
Vector Addition Performance

The graph illustrates the performance of vector addition on different architectures:
- **CPU** line
- **GPU** line
- **GPU+Memcpy** line

The x-axis represents the vector size, ranging from 0 to 1,000,000. The y-axis represents the time taken in seconds, ranging from 0 to 0.014. The CPU line shows a steady increase in time with vector size, while the GPU and GPU+Memcpy lines show a sharp increase, especially for larger vector sizes.
How many threads: GPU Occupancy

- CPU threads share resources by time multiplexing
  - One thread owns all CPU resources (registers, etc.) for its time slice
  - Context-switches are performed by OS
- GPU threads *do not share* resources
  - Own fixed partition of resources for entire lifetime of thread
  - Context-switches are performed by hardware every few cycles
- Changing number of threads changes *utilization* of resources
### GPU Resources per SM (NVIDIA Kepler)

<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>

- Every block consumes:
  - $T$ threads
  - $T \times R$ registers where $R$ is registers per thread
  - 1 block
  - $SM$ shared memory per block (optional)

- The resource that gets exhausted first determines occupancy and residency
  - *Occupancy*: number of hardware threads utilized
  - *Residency*: number of hardware blocks utilized
GPU Occupancy: Example 1

```
kernelf他已经2048, 32>>>()
```

- \( T = 32 \)
  - thread limit \( 2048/32 = 64 \) thread blocks
- \( R = 100 \ (100 \times 32 = 3200 \) per thread block)  
  - register limit \( 65536/3200 = 20 \) thread blocks
- \( SM = 1K \)
  - SM limit \( 48K/1K = 48 \) thread blocks

- Limiting resource: thread blocks (16)
- Residency: 16
- Occupancy: \( (16 \times 32)/2048 = 25\% \)
GPU Occupancy: Example 2

kernel<<<2048, 64>>>()

- \( T = 64 \)
  - thread limit \( \frac{2048}{64} = 32 \) thread blocks
- \( R = 100 \) (\( 100 \times 64 = 6400 \) per thread block)
  - register limit \( \frac{65536}{6400} = ? \) thread blocks
- \( SM = 1K \)
  - SM limit \( \frac{48K}{1K} = 48 \) thread blocks

- Limiting resource: ?
- Residency: ?
- Occupancy: \( (? \times 64) / 2048 = ? \% \)
How many threads?

- Try to maximize utilization (NVIDIA Manual)
- Is there a better strategy?
  - See Volkov, V., "Better Performance at Lower Occupancy", GTC 2010
Outline

Introduction to CUDA

Basic Performance

Memory Performance
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, ...)

Data Layout for GPU programs (SoA)

```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, ...)
```
AoS vs SoA for GPU programs

- Array of Structures
- Structure of Arrays
- Which is better for CPU?
- Which is better for GPU?
• \( p[i].x \) memory bandwidth utilization?
• \(p.x[i]\) memory bandwidth utilization?
AoS vs SoA Performance

The graph shows the comparison of AoS (Array of Structures) and SoA (Single Object Array) performance as a function of array size. The x-axis represents the array size, while the y-axis shows the runtime in milliseconds. The graph indicates that AoS generally has a lower runtime compared to SoA, especially as the array size increases. The blue line represents AoS, and the magenta line represents SoA. The data points suggest that AoS is more efficient for larger array sizes.
AoS vs SoA: Number of Memory Transactions
Assigning Work to Threads

Blocked:

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

Interleaved:

\[
\begin{align*}
\text{start} &= \text{tid}; \\
\text{for}(i = \text{start}; i < \text{N}; i += \text{nthreads}) \\
    &\quad a[i] = b[i] + c[i]
\end{align*}
\]

Which, if any, is faster?
Blocking vs Interleaved
Exploiting Spatial Locality: Texture Caches

- 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
  - i.e. texture caches are not coherent with GPU memory
- Easiest way to use textures:
  - const __restrict__ *
  - Compiler will automatically use texture cache for marked arrays
“Shared Memory” is on-chip software-managed cache, also known as a scratchpad

- 48K maximum size
- Partitioned among thread blocks
- __shared__ qualifier places variables 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);
```
__shared__ float c_sub[BLOCKSIZE][BLOCKSIZE];

// calculate c_sub
__syncthreads();

// write out c_sub to memory
Constant Data Cache

- 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
  - 2dconv: convolution matrix entries
Summary of Memory 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