GPU programming: Code optimization part 1

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
Outline

- Analytical performance modeling
- Optimizing host-device data transfers
- Optimizing for memory locality
  - Matrix multiplication example
- Optimizing memory access patterns
  - Memory access patterns
  - Global memory optimization
First-order performance model

Questions you should ask yourself, before starting to code or optimize

- Will my code run faster on the GPU?
- Is my existing code running as fast as it should?
- Is performance limited by computations or memory bandwidth?

Pen-and-pencil calculations can (often) answer such questions
Optimistic evaluation: upper bound on performance
Assume perfect overlap of computations and memory accesses

- **Memory accesses:** *bytes*
  - Only external memory, not caches or registers

- **Computations:** *flops*
  - Only “useful” computations (usually floating-point) not address calculations, loop iterators..

- **Arithmetic intensity:** *flops / bytes*
  - = computations / memory accesses
  - Property of the code

- **Arithmetic throughput:** *flops / s*
  - Property of code + architecture
The roofline model

- How much performance can I get for a given arithmetic intensity?
  - Upper bound on arithmetic throughput, as a function of arithmetic intensity
  - Property of the architecture

Building the machine model

- Compute or measure:
  - Peak memory throughput: \( \text{GTX 980: 224 GB/s} \)
  - Ideal arithmetic intensity = peak compute throughput / mem throughput
    \[ \text{GTX 980: } \frac{6412 \text{ (Gflops/s)}}{224 \text{ (GB/s)}} = 28.6 \text{ flops/byte} \]
    \[ \times 4 \text{ (B/flop)} = 114 \text{ (dimensionless)} \]
  - Beware of units: float=4B, double=8B!

- Achievable peaks may be lower than theoretical peaks
  - Lower curves when adding realistic constraints
Using the model

- Compute arithmetic intensity, measure performance of program
- Identify bottleneck: memory or computation
- Take optimization decision

**Arithmetic intensity**
(flops/byte)

**Arithmetic throughput**
(Gflops/s)

- Optimize memory accesses
- Optimize computation
- Reuse data

**Measured performance**

**Arithmetic intensity**
(flops/byte)
Example: dot product

\[
\text{for } i = 1 \text{ to } n \\
\quad r += a[i] \times b[i]
\]

- How many computations?
- How many memory accesses?
- Arithmetic intensity?
- Compute-bound or memory-bound?
- How many Gflop/s on a GTX 980 GPU?
  - With data in GPU memory?
  - With data in CPU memory?
- How many Gflop/s on an i7 4790 CPU?

GTX 980: 4612 Gflops/s, 224 GB/s
i7 4790: 460 Gflops/s, 25.6 GB/s
PCIe link: 16 GB/s
Example: dot product

```c
for i = 1 to n
    r += a[i] * b[i]
```

- How many computations? → 2 n flops
- How many memory accesses? → 2 n words
- Arithmetic intensity? → 1 flop/word = 0.25 flop/B
- Compute-bound or memory-bound? → Highly memory-bound

- How many Gflop/s on a GTX 980 GPU?
  - With data in GPU memory? 224 GB/s × 0.25 flop/B → 56 Gflops/s
  - With data in CPU memory? 16 GB/s × 0.25 flop/B → 4 Gflops/s

- How many Gflop/s on an i7 4790 CPU?
  25.6 GB/s × 0.25 flop/B → 6.4 Gflops/s

Conclusion: don't bother porting to GPU!
Outline

- Analytical performance modeling
- Optimizing host-device data transfers
- Optimizing for memory locality
  - Matrix multiplication example
- Optimizing memory access patterns
  - Memory access patterns
  - Global memory optimization
Asynchronous transfers

- Overlap CPU work with GPU work

Can we do better?
Streams: pipelining commands

- **Streams**, aka *Command queues* in OpenCL
  - Commands from the same stream run in-order
  - Commands from different streams run out-of-order
**Streams: benefits**

- Overlap CPU-GPU communication and computation: Direct Memory Access (DMA) copy engine runs CPU-GPU memory transfers in background
  - Requires page-locked memory
  - Some Tesla GPUs have 2 DMA engines: simultaneous send and receive

- Concurrent kernel execution
  - Start next kernel before previous kernel finishes
  - Reduces loss due to load imbalance

Example

```
Kernel<<<5,,,a>>>
Kernel<<<4,,,b>>>
```

Serial kernel execution

```
<table>
<thead>
<tr>
<th>a block 0</th>
<th>a 3</th>
<th>b 0</th>
<th>b 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 4</td>
<td>b 1</td>
<td></td>
</tr>
<tr>
<td>a 2</td>
<td></td>
<td>b 2</td>
<td></td>
</tr>
</tbody>
</table>
```

Concurrent kernel execution

```
<table>
<thead>
<tr>
<th>a block 0</th>
<th>a 3</th>
<th>b 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
<td>a 4</td>
<td>b 1</td>
</tr>
<tr>
<td>a 2</td>
<td>b 0</td>
<td>b 3</td>
</tr>
</tbody>
</table>
```
Page-locked memory

- By default, allocated memory is *pageable*
  - Can be swapped out to disk, moved by the OS...
- DMA transfers are only safe on *page-locked* memory
  - Fixed virtual → physical mapping
  - cudaMemcpy needs an intermediate copy: slower, synchronous only
- cudaMemcpyHost allocates page-locked memory
  - Mandatory when using streams
- Warning: page-locked memory is a limited resource!
Streams: example

- Send data, execute, receive data

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}

for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```
Events

- Schedule synchronization of one stream with another
  - Specifies dependencies between tasks

```c
cudaEvent_t e;
cudaEventCreate(&e);
kernel1<<<,,,a>>>();
cudaEventRecord(e, a);
cudaStreamWaitEvent(b, e);
kernel2<<<,,,b>>>();
cudaEventDestroy(e);
```

- Measure timing

```c
cudaEventRecord(start, 0);
kernel<<<>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```
Scheduling data dependency graphs

- With streams and events, we can express task dependency graphs
  - Equivalent to threads and events (e.g. semaphores) on CPU
- Example:
  - 2 GPU streams: a  b
  - and 1 CPU thread:     
  - Where should we place events?
kernel1<<<,,,a>>>();
cudaEventRecord(e1, a);

kernel2<<<,,,b>>>();
cudaStreamWaitEvent(b, e1);
kernel3<<<,,,b>>>();
cudaEventRecord(e2, b);

kernel5<<<,,,a>>>();
cudaEventRecord(e3, a);
cudaEventSynchronize(e2);

CPU code

cudaStreamWaitEvent(b, e3);
kernel4<<<,,,b>>>();
Outline

- Analytical performance modeling
- Optimizing host-device data transfers
- Optimizing for memory locality
  - Matrix multiplication example
- Optimizing memory access patterns
  - Memory access patterns
  - Global memory optimization
**Classic example: matrix multiplication**

- Naive algorithm

```plaintext
for i = 0 to n-1
  for j = 0 to n-1
    for k = 0 to n-1
      C[i,j]+=A[i,k]*B[k,j]
```

- Arithmetic intensity: 1:1  :(

![Diagrams](image-url)
Reusing inputs

- Move loop on $k$ up

```plaintext
for k = 0 to n-1
  for i = 0 to n-1
    for j = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
```

- Enable data reuse on inputs
- But need to keep all matrix C in registers!
With tiling

- Block loops on i and j

\[
\begin{align*}
&\text{for } i = 0 \text{ to } n-1 \text{ step 16} \\
&\quad \text{for } j = 0 \text{ to } n-1 \text{ step 16} \\
&\quad \quad \text{for } k = 0 \text{ to } n-1 \\
&\quad \quad \quad \text{for } i2 = i \text{ to } i+15 \\
&\quad \quad \quad \quad \text{for } j2 = j \text{ to } j+15 \\
&\quad \quad \quad \quad \quad \quad C[i2,j2] += A[i2,k] \times B[k,j2]
\end{align*}
\]

- For one block: product between horizontal panel of A and vertical panel of B
With more tiling

- Block loop on k

```plaintext
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    for k = 0 to n-1 step 16
      for k2 = k to k+15
        for i2 = i to i+15
          for j2 = j to j+15
            C[i2,j2] += A[i2,k] * B[k,j2]
```

![Diagram showing the block loop on k]
for i = 0 to n-1 step 16
  for j = 0 to n-1 step 16
    c = {0}
    for k = 0 to n-1 step 16
      a = A[i..i+15,k..k+15]
      b = B[k..k+15,j..j+15]
      for k2 = 0 to 15
        for i2 = 0 to 15
          for j2 = 0 to 15
            c[i2,j2]+=a[i2,k2]*b[k2,j2]
  C[i..i+15,j..j+15] = c
Breaking into two levels

- Run loops on i, j, i2, j2 in parallel

```
for // i = 0 to n-1 step 16
  for // j = 0 to n-1 step 16
    c = {0}
    for k = 0 to n-1 step 16
      a = A[i..i+15,k..k+15]
      b = B[k..k+15,j..j+15]
      for k2 = 0 to 15
        for // i2 = 0 to 15
          for // j2 = 0 to 15
            c[i2,j2] += a[i2,k2] * b[k2,j2]
      C[i..i+15,j..j+15] = c
```

- Let's focus on threads
Level 1: SIMD (PRAM-style) version

- Each processor has ID \((x,y)\)
  - Loops on \(i2, j2\) are implicit

\[
\begin{align*}
c[x,y] &= 0 \\
\text{for } k = 0 \text{ to } n-1 \text{ step } 16 \\
a[x,y] &= A[i+x,k+y] \\
b[x,y] &= B[k+x,j+y] \\
\text{for } k2 = 0 \text{ to } 15 \\
c[x,y] &= a[x,k2] \times b[k2,y] \\
\text{C}[i+x,j+y] &= c[x,y]
\end{align*}
\]

- Load submatrices \(a\) and \(b\)
- Multiply submatrices \(a \times b\)
- Store submatrix \(c\)
- Private writes: no conflict
- Read from other processors

- How to translate to SPMD (BSP-style)?
SPMD version

- Place synchronization barriers

```plaintext
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```

- Why do we need the second barrier?
Data allocation

- 3 memory spaces: Global, Shared, Local

- Where should we put: A, B, C, a, b, c?

\[
\begin{align*}
c[x,y] &= 0 \\
\text{for } k &= 0 \text{ to } n-1 \text{ step 16} \\
a[x,y] &= A[i+x,k+y] \\
b[x,y] &= B[k+x,j+y] \\
\text{Barrier} \\
\text{for } k2 &= 0 \text{ to } 15 \\
c[x,y] &= c[x,y] + a[x,k2] \times b[k2,y] \\
\text{Barrier} \\
C[i+x,j+y] &= c[x,y]
\end{align*}
\]