GPU programming: Code optimization part 2

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

- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Example: back to matrix multiplication
- Workload partitioning
  - Balancing ILP and TLP
- Instruction-level optimization
  - Warp divergence
Parallel regularity

- Similarity in behavior between threads

<table>
<thead>
<tr>
<th>Control regularity</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i=17</td>
<td>i=21</td>
</tr>
<tr>
<td>2</td>
<td>i=17</td>
<td>i=4</td>
</tr>
<tr>
<td>3</td>
<td>i=17</td>
<td>i=17</td>
</tr>
<tr>
<td>4</td>
<td>i=17</td>
<td>i=2</td>
</tr>
</tbody>
</table>

```
switch(i) {
    case 2:...
    case 17:....
    case 21:....
}
```

<table>
<thead>
<tr>
<th>Memory regularity</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Memory</td>
</tr>
<tr>
<td>load A[9]</td>
<td></td>
</tr>
<tr>
<td>load A[10]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data regularity</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a=32</td>
<td>a=17</td>
</tr>
<tr>
<td>a=32</td>
<td>a=-5</td>
</tr>
<tr>
<td>a=32</td>
<td>a=11</td>
</tr>
<tr>
<td>a=32</td>
<td>a=42</td>
</tr>
<tr>
<td>b=52</td>
<td>b=15</td>
</tr>
<tr>
<td>b=52</td>
<td>b=0</td>
</tr>
<tr>
<td>b=52</td>
<td>b=-2</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
</tr>
</tbody>
</table>

r=a*b
Memory access patterns

In traditional vector processing

Easy
Registers
Memory
Scalar load & broadcast
Reduction & scalar store

Hard
Registers
Memory
(Non-unit) strided load
(Non-unit) strided store

Hardest
Registers
Memory
Gather
Scatter

On GPUs
- Every load is a gather, every store is a scatter
Breakdown of memory access patterns

- Vast majority: uniform or unit-strided
  - And even aligned vectors

“*In making a design trade-off, favor the frequent case over the infrequent case.*”

Memory coalescing

- In hardware: compare the address of each vector element
- Coalesce memory accesses that fall within the same segment

→ One transaction

Unit-strided requests

Irregular requests

→ Multiple transactions

- Dynamically detects parallel memory regularity
Coalescing concurrent requests

- Unit-strided detection (NVIDIA CC 1.0-1.1 coalescing)
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

- Minimal coverage (NVIDIA CC 1.2 coalescing)
  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

Impact on work distribution, data structures?
Consequences: threading granularity

- **Coarse-grained threading**
  - **Decouple** tasks to reduce **conflicts** and inter-thread communication
  - e.g. MPI, OpenMP

- **Fine-grained threading**
  - **Interleave** tasks
  - Exhibit **locality**: neighbor threads share memory
  - Exhibit **regularity**: neighbor threads have a similar behavior
  - e.g. CUDA, OpenCL
Array of structures (AoS)

- Programmer-friendly memory layout
  - Group data logically
- Memory accesses not coalesced
  - Bad performance on GPU

- Need to rethink data structures for fine-grained threading

```c
struct Pixel {
    float r, g, b;
};
Pixel image_AoS[480][640];

kernel void luminance(Pixel img[][], float luma[][]) {
    int x=tid.x; int y=tid.y;
    luma[y][x] = .59*img[y][x].r
                + .11*img[y][x].g
                + .30*img[y][x].b;
}
```
Structure of Arrays (SoA)

- Transpose the data structure
  - Group together similar data for different threads

- Benefits from memory coalescing
  - Best performance on GPU

```cpp
struct Image {
    float R[480][640];
    float G[480][640];
    float B[480][640];
};

Image image_SoA;

kernel void luminance(Image img, float luma[][[]]) {
    int x = tid.x; int y = tid.y;
    luma[y][x] = .59*img.R[y][x] + .11*img.G[y][x] + .30*img.B[y][x];
}
```
Outline

- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Example: back to matrix multiplication
- Workload partitioning
  - Balancing ILP and TLP
- Instruction-level optimization
  - Warp divergence
Vector loads

- We can load more data at once with vector types
  - float2, float4, int2, int4...
  - More memory parallelism
  - Allows to reach peak throughput with fewer threads

Multiple outstanding loads

- Multiple independent loads from the same thread can be pipelined
  - More memory parallelism
  - Peak throughput with yet fewer threads

```c
__global__ void luminance(Image img,
    float luma[][[]]) {
    int x=threadIdx.x, y=threadIdx.y;
luma[y][x]=.59*img.R[y][x]
    + .11*img.G[y][x]
    + .30*img.B[y][x];
}
```
Global memory accesses are the most expensive

- Focus on optimizing global memory accesses

Strategy: use shared memory as a temporary buffer

1. Load with regular accesses
2. Read and write shared memory with original pattern
3. Store back to global memory with regular accesses
Example: matrix transpose

- $B = A^T$
- Naive algorithms
  - Option 1
    
    Thread $i, j$:
    
    $B[j, i] = A[i, j]$
  
  - Option 2
    
    Thread $i, j$:
    
    $B[i, j] = A[j, i]$

- Which one is better?
  - What is the problem?
Example: matrix transpose

- $B = A^T$
- Naive algorithms
  - Option 1
    - Coalesced
  - Option 2
    - Non-coalesced
- Both are equally bad
  - Access to one array is non-coalesced
Matrix transpose using shared memory

- Split matrices in blocks
- Load the block in shared memory
- Transpose in shared memory
- Write the block back

Example with $16 \times 16$ blocks

Block $bx, by$, Thread $tx, ty$:

\[
a[ty, tx] = A[by \times 16 + ty, by \times 16 + tx]
\]

Syncthreads

\[
b[ty, tx] = a[tx, ty]
\]

Syncthreads

\[
B[by \times 16 + ty, bx \times 16 + tx] = b[ty, tx]
\]
Objection

- Isn't it just moving the problem to shared memory?
- Yes: shared memory has access restrictions too
- But
  - Shared memory is much faster, even for irregular accesses
  - We can optimize shared memory access patterns too
Outline

• Memory optimization
  ♦ Memory access patterns
  ♦ Global memory optimization
  ♦ Shared memory optimization
  ♦ Example: back to matrix multiplication

• Workload partitioning
  ♦ Balancing ILP and TLP

• Instruction-level optimization
  ♦ Warp divergence
• Inside each SM, shared memory is distributed between multiple banks
  • 16 or 32 banks
Shared memory bank assignment

- Interleaved on a word-by-word basis: Modulo placement of data

```
0
0xffff
```

- Actually 16 (or 32) banks

```
Shared memory address space
```

```
Bank 0  Bank 1  Bank 2  Bank 3
```

32-bit
Shared memory: the good

- Threads access contiguous locations: no conflict
  - All threads can be served concurrently
Shared memory: the bad

- Threads access random locations: some conflicts
  - Some threads have to wait for a bank
Shared memory: the ugly

- Threads access locations spaced by 16: systematic conflict
  - All threads have to wait for the same bank
Example: matrix transpose

- Where are bank conflicts?

Block bx, by, Thread tx, ty:
- \( a[ty*16+tx] = A[by*16+ty, by*16+tx] \)
  - `Syncthreads`
- \( b[ty*16+tx] = a[tx*16+ty] \)
  - `Syncthreads`
- \( B[by*16+ty, bx*16+tx] = b[ty*16+tx] \)
Example: matrix transpose

- Where are bank conflicts?

Block bx, by, Thread tx, ty:
\[ a[ty*16+tx] = A[by*16+ty, by*16+tx] \]
Sync threads
\[ b[ty*16+tx] = a[tx*16+ty] \]
Sync threads
\[ B[by*16+ty, bx*16+tx] = b[ty*16+tx] \]

- How to avoid them?
Remapping data

- Solution 1: pad with empty cells

Block bx, by, Thread tx, ty:
\[ a[ty*17+tx] = A[by*16+ty, by*16+tx] \]
Sync threads
\[ b[ty*16+tx] = a[tx*17+ty] \]
Sync threads
\[ B[by*16+ty, bx*16+tx] = b[ty*17+tx] \]

- No bank conflicts
- Memory overhead
Remapping data

- Solution 2: different mapping function
  - Example: map \([y, x]\) to \(y \times 16 + (x + y \mod 16)\)
  - Or \(y \times 16 + (x ^ y)\)

Block \(bx, by\), Thread \(tx, ty\):
- \(a[ty \times 16 + (tx + ty) \mod 16] = A[by \times 16 + ty, by \times 16 + tx]\)
- Syncthreads
- \(b[ty \times 16 + tx] = a[tx \times 16 + (ty + tx) \mod 16]\)
- Syncthreads
- \(B[by \times 16 + ty, bx \times 16 + tx] = b[ty \times 17 + tx]\)

- No bank conflicts
- No memory overhead
Recap

- Overlap long-latency communications with computations
- Avoid global accesses when you can
  - Reuse data to get enough arithmetic intensity
  - Use registers and shared memory whenever possible
- Make consecutive threads access contiguous data
  - Stage data in shared memory if needed
- Avoid bank conflicts in shared memory
- Express locality and regularity
Outline

- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Example: back to matrix multiplication
- Workload partitioning
  - Balancing ILP and TLP
- Instruction-level optimization
  - Warp divergence
Example: back to matrix multiplication

- On a block of 256 threads

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T16</th>
<th>T17</th>
<th>T255</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

\[ c = 0 \]
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

\[ c += a[x,k2]*b[k2,y] \]
Barrier

\[ C[i+x,j+y] = c \]

- Which accesses are coalesced?
- Are there bank conflicts?
Example: back to matrix multiplication

- On a block of 256 threads

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T16</th>
<th>T17</th>
<th>T255</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

\[ c = 0 \]
for \( k = 0 \) to \( n-1 \) step 16
\[
\begin{align*}
a[x,y] &= A[i+x,k+y] \\
b[x,y] &= B[k+x,j+y] \\
\end{align*}
\]
Barrier
for \( k2 = 0 \) to 15
\[
\begin{align*}
c &== a[x,k2] \times b[k2,y] \\
\end{align*}
\]
Barrier
\[
C[i+x,j+y] = c
\]
No coalesced access
Massive bank conflicts

```plaintext
c = 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 += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c
```

Can we improve it?
Memory optimization

- Exchange x and y

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

- Success!
- Now can we improve memory parallelism?
Outline

- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Example: back to matrix multiplication

- Workload partitioning
  - Balancing ILP and TLP

- Instruction-level optimization
  - Warp divergence
Workload partitioning

How to choose grid dimensions?

- **Number of blocks per grid**
  - Linear with data size, or constant
  - Min: at least number of SMs * blocks per SM
  - No max in practice

- **Number of threads per block**
  - Constant: should not depend on dataset size
  - Max: hardware limitation, 512 or 1024 threads
  - Min: size of a warp: 32 threads

- **Iterations per thread**
  - Constant or variable
  - Min: enough to amortize thread creation overhead
  - No max, but shorter-lived threads reduce load imbalance
Multiple grid/block dimensions

- Grid and block size are of type `dim3`
  - Support up to 3 dimensions
    ```cpp
    dim3 dimBlock(tx, ty, tz);
    dim3 dimGrid(bx, by, bz);
    my_kernel<<<dimGrid, dimBlock>>>(arguments);
    ```
  - Implicit cast from int to `dim3`
    - Y and Z sizes are 1 by default
- On device side, `threadIdx`, `blockDim`, `blockIdx`, `gridDim` are also of type `dim3`
  - Access members with `.x`, `.y`, `.z`
- Some early GPUs (CC 1.x) only support 2-D grids
Occupancy metric

- Threads per SM / max threads per SM
- Resource usage may cause non-ideal occupancy
  - Register usage
  - Shared memory usage
  - Non-dividable block size

Available shared memory: 16KB
Usage: 12KB/block
→ Only 1 block / SM

Available registers: 32768
Usage: 64 registers/thread, blocks of 256 threads
→ Only 2 blocks / SM

Max threads/SM: 768 threads
Block size: 512 threads
→ Only 1 block / SM
Could run 3 blocks of 256 threads
GPU: on-chip memory

- Conventional wisdom
  - Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:

- But... if we include registers:

<table>
<thead>
<tr>
<th>GPU</th>
<th>Register files + caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GM204 GPU</td>
<td>8.3 MB</td>
</tr>
<tr>
<td>AMD Hawaii GPU</td>
<td>15.8 MB</td>
</tr>
<tr>
<td>Intel Core i7 CPU</td>
<td>9.3 MB</td>
</tr>
</tbody>
</table>

- GPU/accelerator internal memory exceeds desktop CPUs
How many threads?

- As many as possible (maximize occupancy)?
  - Maximal data-parallelism
    - Latency hiding
  - Locality
    - Store private data of each thread
  - Thread management overhead
    - Initialization, redundant operations
- Trade-off between parallelism and memory locality
Multiple elements per thread

- Block size $(16, 16) \rightarrow (8, 16)$
- 2 elements per thread: $(x, y)$ and $(x+8, y)$

\[
\begin{align*}
    c[0] &= 0 \\
    c[1] &= 0 \\
    \text{for } k = 0 \text{ to } n-1 \text{ step 16} & \quad \text{More outstanding loads} \\
    a[y,x] &= A[i+y,k+x] \\
    b[y,x] &= B[k+y,j+x] \\
    a[y+8,x] &= A[i+y+8,k+x] \\
    b[y+8,x] &= B[k+y+8,j+x] \\
    \text{Barrier} & \\
    \text{for } k2 = 0 \text{ to } 15 & \\
    c[0] &= a[y,k2] \cdot b[k2,x] \\
    c[1] &= a[y+8,k2] \cdot b[k2,x] \\
    \text{Barrier} & \\
    C[i+y,j+x] &= c[0] \\
    C[i+y+8,j+x] &= c[1]
\end{align*}
\]

- What about shared memory?
Data reuse

- Share reads to submatrix $b$
  - Fewer shared memory accesses
  - Exchange data through registers

```plaintext
c[0] = 0
c[1] = 0
for k = 0 to n-1 step 16
  a[y,x] = A[i+y,k+x]
  b[y,x] = B[k+y,j+x]
  a[y+8,x] = A[i+y+8,k+x]
  b[y+8,x] = B[k+y+8,j+x]
  Barrier
  for k2 = 0 to 15
    bl = b[k2,x]
    c[0] += a[y,k2]*bl
    c[1] += a[y+8,k2]*bl
  Barrier
  C[i+y,j+x] = c[0]
  C[i+y+8,j+x] = c[1]
```

- Improves register usage too. Why?
True story: SGEMM from CUBLAS 1.1


- 512 threads / CTA, 15 registers / thread
- 9 registers / 15 contain redundant data
- Only 2 registers really needed
Fewer threads, more computations

- New SGEMM in CUBLAS 2.0
  - 8 elements computed / thread
  - Unrolled loops
  - Less traffic through shared memory, more through registers

- Overhead amortized
  - 1920 registers vs. 7680 for the same amount of work
  - Works for redundant computations too

- Instruction-level parallelism is still relevant
Re-expressing parallelism

- Converting types of parallelism
  - ILP
  - TLP
  - DLP

- General strategy
  - Design phase: focus on thread-level parallelism
  - Optimization phase: convert TLP to Instruction-level or Data-level parallelism
Outline

- Memory optimization
  - Memory access patterns
  - Global memory optimization
  - Shared memory optimization
  - Example: back to matrix multiplication

- Workload partitioning
  - Balancing ILP and TLP

- Instruction-level optimization
  - A few device code features
  - Warp divergence
Device functions

- Kernel can call functions
- Need to be marked for GPU compilation
  
  ```c
  __device__ int foo(int i) {
  }
  ```

- A function can be compiled for both host and device
  
  ```c
  __host__ __device__ int bar(int i) {
  }
  ```

- Device functions can call device functions
  - Older GPUs do not support recursion
Local memory

- Registers are fast but
  - Limited in size
  - Not addressable

- Local memory used for
  - Local variables that do not fit in registers (*register spilling*)
  - Local arrays accessed with indirection

```c
int a[17];
b = a[i];
```

- **Warning**: local is a misnomer!
  - Physically, local memory usually goes off-chip
  - About same performance as coalesced access to global memory
Loop unrolling

- Can improve performance
  - Amortizes loop overhead over several iterations
  - May allow constant propagation, common sub-expression elimination...

- Unrolling is **necessary** to keep arrays in registers

```c
int a[4];
for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
}
```

Not unrolled

```
int a[4];
for(int i = 0; i < 4; i++) {
    a[0] = 3 * 0;
    a[1] = 3 * 1;
    a[2] = 3 * 2;
    a[3] = 3 * 3;
}
```

Unrolled

Indirect addressing: 
a in local memory

Static addressing: 
a in registers

- The compiler can unroll for you
  ```c
  #pragma unroll
  for(int i = 0; i < 4; i++) {
      a[i] = 3 * i;
  }
  ```
  Trivial computations: optimized away
Warp-based execution

- Threads in a warp run in lockstep
- On NVIDIA architectures, warp is 32 threads
- A block is made of warps (warps do not cross block boundaries)
  - Block size multiple of 32 for best performance
Branch divergence

- Conditional block
  ```c
  if(c) {
      // A
  }
  else {
      // B
  }
  ```

- All threads of a warp take the same path
  ```
  T_0 T_1 T_2 T_3
  A A A A
  \[\text{Warp 0}\]
  
  T_4 T_5 T_6 T_7
  B B B B
  \[\text{Warp 1}\]
  ```

  With imaginary 4-thread warps
Branch divergence

- Conditional block

```cpp
if(c) {
    // A
}
else {
    // B
}
```

- Threads in a warp take different paths

- Warps have to go through both A and B: lower performance
Avoiding branch divergence

- Hoist identical computations and memory accesses outside conditional blocks

```c
if(tid % 2) {
    s += 1.0f/tid;
} else {
    s -= 1.0f/tid;
}

float t = 1.0f/tid;

if(tid % 2) {
    s += t;
} else {
    s -= t;
}
```

- When possible, re-schedule work to make non-divergent warps

```c
// Compute 2 values per thread
int i = 2 * tid;
s += 1.0f/i - 1.0f/(i+1);
```

- Q: What if I use C's ternary operator (?:) instead of if?
Answer: ternary operator or predication

- Run both branches and select: $R = c \ ? \ A : B$;
  - No more divergence?
- All threads have to take both paths
  No matter whether the condition is divergent or not

Does **not** solve divergence: we lose in all cases!

- Only benefit: fewer instructions
  - May be faster for short, often-divergent branches
- Compiler will choose automatically when to use predication
  - Advice: keep the code readable, let the compiler optimize
Recap

- Beware of local arrays
  use static indices and loop unrolling

- Keep in mind branch divergence when writing algorithm
  but do not end up in managing divergence yourself
Takeaway

- Distribute work and data
  - Favor SoA
  - Favor locality and regularity
  - Use common sense (avoid extraneous copies or indirections)
- More threads ≠ higher performance
  - Saturate instruction-level parallelism first (almost free)
  - Complete with data parallelism (expensive in terms of locality)
- Usual advice applies
  - First write correct code
  - Profile
  - Optimize
  - Repeat
References and further reading/watching

- CUDA C Programming Guide