Device Memories and Matrix Multiplication

1. Device Memories
   - global, constant, and shared memories
   - CUDA variable type qualifiers

2. Matrix Multiplication
   - an application of tiling
   - running `matrixMul` in the GPU Computing SDK
   - the kernel of `matrixMul`
Device Memories and Matrix Multiplication

1. Device Memories
   - global, constant, and shared memories
   - CUDA variable type qualifiers

2. Matrix Multiplication
   - an application of tiling
   - running `matrixMul` in the GPU Computing SDK
   - the kernel of `matrixMul`
Before we launch a kernel, we have
- to allocate memory on the device,
- to transfer data from the host to the device.

By default, memory on the device is *global memory*. Global memory on a GPU plays the same role as the Random Access Memory on a CPU.

In addition to global memory, we distinguish between
- registers for storing local variables,
- shared memory for all threads in a block,
- constant memory for all blocks on a grid.
compute to global memory access (CGMA) ratio

The importance of understanding different memories is in the calculation of the expected performance level of kernel code.

**Definition (CGMA ratio)**

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

If the CGMA ratio is 1.0, then the memory clock rate determines the upper limit for the performance.

While memory bandwidth on a GPU is superior to that of a CPU, we will miss the theoretical peak performance by a factor of ten.
CUDA device memory types

- **grid**
- **block (0,0)**
  - shared memory
  - registers
  - thread(0,0)
  - thread(1,0)
- **block (1,0)**
  - shared memory
  - registers
  - thread(0,0)
  - thread(1,0)
- **host**
  - global memory
  - constant memory
**registers**

Registers are allocated to individual threads. Each thread can access only its own registers.

A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

Number of 32-bit registers available per block:
- 8,192 on the GeForce 9400M,
- 32,768 on the Tesla C2050/C2070,
- 65,536 on the Tesla K20C, the P100 and V100.

A typical CUDA kernel may launch thousands of threads. However, having too many local variables in a kernel function may prevent all blocks from running in parallel.
shared memory

Like registers, shared memory is an on-chip memory.

Variables residing in registers and shared memory can be accessed at very high speed in a highly parallel manner.

Unlike registers, which are private to each thread, all threads in the same block have access to shared memory.

Amount of shared memory per block:

- 16,384 bytes on the GeForce 9400M,
- 49,152 bytes on the Tesla C2050/C2070,
- 49,152 bytes on the Tesla K20c, the P100 and V100.
constant, global, and cache memory

The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

Global memory is similar to RAM on the CPU.

<table>
<thead>
<tr>
<th>GPU</th>
<th>constant</th>
<th>global</th>
<th>L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce 9400M</td>
<td>65,536 b</td>
<td>254 Mb</td>
<td></td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>65,536 b</td>
<td>2,687 Mb</td>
<td>786,432 b</td>
</tr>
<tr>
<td>Tesla K20C</td>
<td>65,536 b</td>
<td>4,800 Mb</td>
<td>1,310,720 b</td>
</tr>
<tr>
<td>Tesla P100</td>
<td>65,536 b</td>
<td>16,276 Mb</td>
<td>4,194,304 b</td>
</tr>
<tr>
<td>Tesla V100</td>
<td>65,536 b</td>
<td>32,505 Mb</td>
<td>6,291,456 b</td>
</tr>
</tbody>
</table>
a quick refresher

copied from the NVIDIA Whitepaper on Kepler GK110
Device Memories and Matrix Multiplication

1. Device Memories
   - global, constant, and shared memories
   - CUDA variable type qualifiers

2. Matrix Multiplication
   - an application of tiling
   - running `matrixMul` in the GPU Computing SDK
   - the kernel of `matrixMul`
variables in memory, scope, and lifetime

Each variable is stored in a particular type of memory, has a scope and a lifetime.

**Scope** is the range of threads that can access the variable.

- If the scope of a variable is a single thread, then a private version of that variable exists for every single thread.
- Each thread can access only its private version of the variable.

**Lifetime** specifies the portion of the duration of the program execution when the variable is available for use.

- If a variable is declared in the kernel function body, then that variable is available for use only by the code of the kernel.
- If the kernel is invoked several times, then the contents of that variable will not be maintained across these invocations.
CUDA variable type qualifiers

We distinguish between five different variable declarations, based on their memory location, scope, and lifetime.

<table>
<thead>
<tr>
<th>variable declaration</th>
<th>memory</th>
<th>scope</th>
<th>lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic variables ≠ arrays</td>
<td>register</td>
<td>thread</td>
<td>kernel</td>
</tr>
<tr>
<td>array variables</td>
<td></td>
<td>thread</td>
<td>kernel</td>
</tr>
<tr>
<td><strong>device</strong>.<strong>shared</strong>.int v</td>
<td></td>
<td>block</td>
<td>kernel</td>
</tr>
<tr>
<td><strong>device</strong>.int v</td>
<td></td>
<td>grid</td>
<td>program</td>
</tr>
<tr>
<td><strong>device</strong>.<strong>constant</strong>.int v</td>
<td></td>
<td>grid</td>
<td>program</td>
</tr>
</tbody>
</table>

The __device__ in front of __shared__ is optional.
Device Memories and Matrix Multiplication

1. Device Memories
   - global, constant, and shared memories
   - CUDA variable type qualifiers

2. Matrix Multiplication
   - an application of tiling
   - running `matrixMul` in the GPU Computing SDK
   - the kernel of `matrixMul`
the CGMA ratio

In our simple matrix-matrix multiplication $C = A \cdot B$, we have the statement

$$C[i] += (*pA++) \times (*pB);$$

where

- $C$ is a float array; and
- $pA$ and $pB$ are pointers to elements in a float array.

For the statement above, the CGMA ratio is 2/3:

- for one addition and one multiplication,
- we have three memory accesses.
an application of tiling

For $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, the product $C = A \cdot B \in \mathbb{R}^{n \times p}$.

Assume that $n$, $m$, and $p$ are multiples of some $w$, e.g.: $w = 8$.

We compute $C$ in tiles of size $w \times w$:

- Every block computes one tile of $C$.
- All threads in one block operate on submatrices:

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}.$$  

- The submatrices $A_{i,k}$ and $B_{k,j}$ are loaded from global memory into shared memory of the block.
matrix multiplication with shared memory

\[
C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}
\]

\[
A \quad B
\]

\[
C
\]

\[
\begin{bmatrix}
A \\
\end{bmatrix}
\begin{bmatrix}
B \\
\end{bmatrix}
\begin{bmatrix}
C \\
\end{bmatrix}
\]

\[
C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}
\]
Device Memories and Matrix Multiplication

1. Device Memories
   - global, constant, and shared memories
   - CUDA variable type qualifiers

2. Matrix Multiplication
   - an application of tiling
   - running `matrixMul` in the GPU Computing SDK
   - the kernel of `matrixMul`
The matrix-matrix multiplication is explained in great detail in the CUDA programming guide.

One of the examples in the GPU Computing SDK is `matrixMul`. We run it on the GeForce 9400M, the Tesla C2050/C2070, the Tesla K20c (on `kepler`), and P100 (on `pascal`).
on the GeForce 9400M

/Developer/GPU Computing/C/bin/darwin/release $ ./matrixMul
[matrixMul] starting...

[ matrixMul ]
./matrixMul
Starting (CUDA and CUBLAS tests)...

Device 0: "GeForce 9400M" with Compute 1.1 capability

Using Matrix Sizes: A(160 x 320), B(160 x 320), C(160 x 320)

Running Kernels...

> CUBLAS 7.2791 GFlop/s, Time = 0.00225 s, Size = 16384000 Ops

> CUDA matrixMul 5.4918 GFlop/s, Time = 0.00298 s, Size = 16384000 Ops,
NumDevsUsed = 1, Workgroup = 256

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results... PASSED
on the Tesla C2050/C2070

/usr/local/cuda/sdk/C/bin/linux/release jan$ ./matrixMul
[matrixMul] starting...
[ matrixMul ]
./matrixMul Starting (CUDA and CUBLAS tests)...

Device 0: "Tesla C2050 / C2070" with Compute 2.0 capability

Using Matrix Sizes: A(640 x 960), B(640 x 640), C(640 x 960)

Running Kernels...

> CUBLAS Throughput = 424.8840 GFlop/s, Time = 0.00185 s, \ Size = 786432000 Ops

> CUDA matrixMul Throughput = 186.7684 GFlop/s, Time = 0.00421 s, \ Size = 786432000 Ops, NumDevsUsed = 1, Workgroup = 1024

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results...
PASSED
on the K20c

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...

GPU Device 0: "Tesla K20c" with compute capability 3.5
MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 246.13 GFlop/s, Time= 0.533 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

Note: For peak performance, please refer to the matrixMulCUBLAS example.

```
$

The theoretical peak performance of the K20c is 1.17 TFlops double precision, and 3.52 TFlops single precision.

The matrices that are multiplied have single float as type.
going for peak performance with CUBLAS

$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
/usr/bin/nvidia-modprobe: unrecognized option: "-u"

GPU Device 0: "Tesla K20c" with compute capability 3.5

MatrixA(320,640), MatrixB(320,640), MatrixC(320,640)
Computing result using CUBLAS...done.
Performance= 1171.83 GFlop/s, Time= 0.112 msec, Size= 131072000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
$

The theoretical peak performance of the K20c is 1.17 TFlops double precision,
and 3.52 TFlops single precision.

The matrices that are multiplied have single float as type.
on the P100

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 1909.26 GFlop/s, Time= 0.069 msec, Size= 131072000 Ops,
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements.
Results may vary when GPU Boost is enabled.
$

The theoretical peak performance (with GPU Boost):
18.7 TFlops (half), 9.3 TFlops (single), 4.7 TFlops (double).
```
running CUBLAS on P100

```
$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS [Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 3089.82 GFlop/s, Time= 0.064 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```

A second run gave the following:

```
Performance= 3106.43 GFlop/s, Time= 0.063 msec, Size= 196608000 Ops
```

For single floats, the theoretical peak performance is 9.3 TFlops.
Device Memories and Matrix Multiplication

### Device Memories
- global, constant, and shared memories
- CUDA variable type qualifiers

### Matrix Multiplication
- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- the kernel of `matrixMul`
the kernel of `matrixMul`

template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;    // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;   // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;
the submatrices

// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
loading and multiplying

// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}
the end of the kernel

// Write the block sub-matrix to device memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

The emphasis in this lecture is on

1. the use of device memories; and
2. data organization (tiling) and transfer.

In the next lecture we will come back to this code, and cover thread scheduling

1. the use of blockIdx; and
2. thread synchronization.
summary and exercises


We covered more of chapter 3 in the book of Kirk & Hwu, and also several concepts explained in chapter 5.

1. Compile the matrixMul of the GPU Computing SDK on your laptop and desktop and run the program.

2. Consider the matrix multiplication code of last lecture and compute the CGMA ratio.

3. Adjust the code for matrix multiplication we discussed last time to use shared memory.