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`
3. CUDA.jl
   - using shared memory

MCS 572 Lecture 20
Introduction to Supercomputing
Jan Verschelde, 11 October 2024

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`
3. CUDA.jl
   - using shared memory

data storage on a graphics card

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.

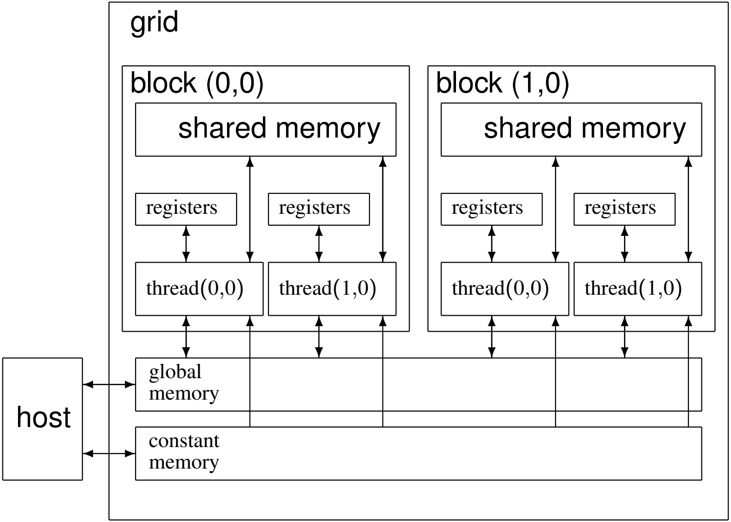
Definition (CGMA ratio)

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



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 K20C, the P100, V100, and A100.

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 byes on the GeForce 9400M,
- 49,152 bytes on the Tesla C2050/C2070,
- 49,152 bytes on the K20c, the P100, V100, and A100.

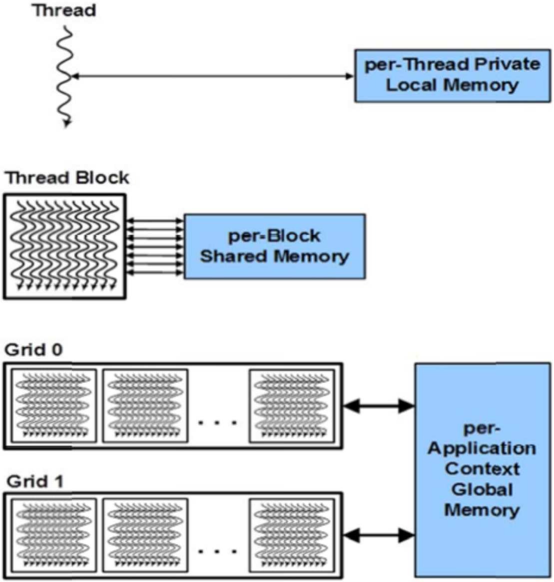
constant, global, and cache memory

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.

GPU	constant	global	L2 cache
GeForce 9400M	65,536 b	254 Mb	
Tesla C2050	65,536 b	2,687 Mb	786,432 b
Kepler K20C	65,536 b	4,800 Mb	1,310,720 b
Pascal P100	65,536 b	16,276 Mb	4,194,304 b
Volta V100	65,536 b	32,505 Mb	6,291,456 b
Ampere A100	65,536 b	81,038 Mb	41,943,040 b

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`

3. CUDA.jl
   - using shared memory

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.

variable declaration	memory	scope	lifetime
atomic variables $\neq$ arrays	register	thread	kernel
array variables	local	thread	kernel
`__device__.__shared__.int v`	shared	block	kernel
`__device__.int v`	global	grid	program
`__device__.__constant__.int v`	constant	grid	program

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`

3. CUDA.jl
   - using shared memory

the CGMA ratio

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

```
C[i] += (*(pA++))*(*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 $w \times w$:

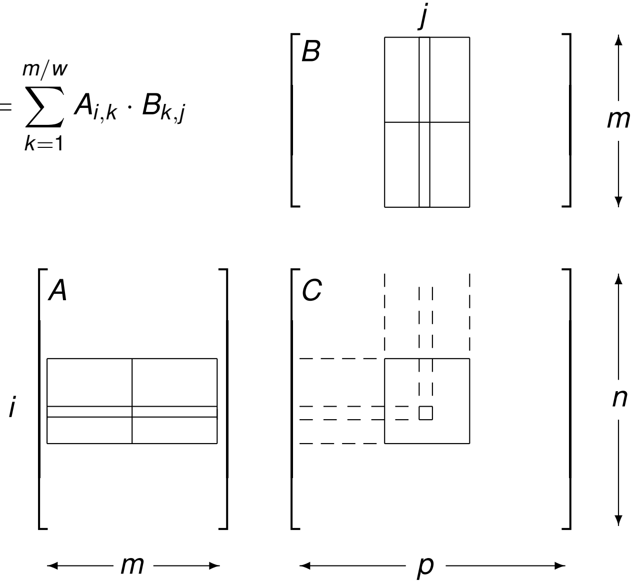
- Every block computes one tile of C.
- All threads in one block operate on submatrices:

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

- 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

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



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`

3. CUDA.jl
   - using shared memory

the `matrixMul` in the GPU Computing SDK

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`), P100 (on `pascal`), and A100 (on `ampere`).

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)

Runing 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)

Runing 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 Kepler 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 TFlop/s double precision, and 3.52 TFlop/s 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 TFlop/s double precision, and 3.52 TFlop/s single precision.

The matrices that are multiplied have single float as type.

on Pascal 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 TFlop/s (half), 9.3 TFlop/s (single), 4.7 TFlop/s (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 TFlop/s.

on Volta V100

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Quadro GV100" with compute capability 7.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 2974.46 GFlop/s, Time= 0.044 msec, Size= 131072000 Ops
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

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

Observe the performance of 2.9 TFlop/s.

The theoretical peak performance (with GPU Boost):
29.6 TFlop/s (half), 14.8 TFlop/s (single), 7.9 TFlop/s (double).

running CUBLAS on V100

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Quadro GV100" with compute capability 7.0

GPU Device 0: "Quadro GV100" with compute capability 7.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 7146.40 GFlop/s, Time= 0.028 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.
[jan@volta release]$ pwd
/usr/local/cuda/samples/bin/x86_64/linux/release
$
```

Observe the performance of 7.1 TFlop/s.
For single floats, the theoretical peak precision is 14.8 Tflop/s with GPU Boost.

on Ampere A100

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Ampere" with compute capability 8.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done

Performance= 4303.49 GFlop/s, Time= 0.030 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.
```

Observe the 4.303 TFlop/S performance.

The theoretical peak performance (with GPU Boost):
78 TFlop/s (half), 19.5 TFlop/s (single), 9.7 TFlop/s (double).

running CUBLAS on A100

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Ampere" with compute capability 8.0

GPU Device 0: "NVIDIA A100 80GB PCIe" with compute capability 8.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 11076.92 GFlop/s, Time= 0.018 msec, Size= 196608000 Op
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.
```

Observe the 11.076 TFlop/s performance.

For single floats, the theoretical peak precision is 17.6 TFlop/s, or 19.5 TFlop/s with GPU Boost.

matrixMul performance evolution

The table below summarizes the experiments, on four GPUs:

GPU units	matrixMul GFlop/s	CUBLAS GFlop/s	peak performance TFlop/s
K20C	264	1,171	3.5
P100	1,909	3,089	9.3
V100	2,974	7,146	14.8
A100	4,303	11,076	19.5

Observe the steady progress of teraflop performance.

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`

3. CUDA.jl
   - using shared memory

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.

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`
3. CUDA.jl
   - using shared memory

compute a dot product with shared memory

adapted from the CUDA.jl documentation

```
using CUDA
"""
    function dot(a,b,c, N, threadsPerBlock, blocksPerGrid)

computes the dot product of two vectors a and b of length N
and places the result in c, using shared memory.
"""
function dot(a,b,c, N, threadsPerBlock, blocksPerGrid)

    # Set up shared memory cache for this current block.
    cache = @cuDynamicSharedMem(Int64, threadsPerBlock)
```

The kernel computes $c_i = \sum_{k=1}^{n} a_k b_k$, where n equals the number of threads per block. The main program adds up the c_i for each block.

use of shared memory in the sum

The `cache` is used in the reduction:

```
i::Int = blockDim().x/2
while i!=0
   if cacheIndex < i
       cache[cacheIndex+1] += cache[cacheIndex+i+1]
   end
   sync_threads()
   i = i/2
end
```

At the end of the kernel:

```
if cacheIndex == 0
    c[blockIdx().x] = cache[1]
end
```

launching the kernel

We start with the dimensions:

```
N::Int64 = 33 * 1024
threadsPerBlock::Int64 = 256
blocksPerGrid::Int64 = min(32,
    (N + threadsPerBlock - 1) / threadsPerBlock)
```

Then comes the setup of the data (omitted).
The launching of the kernels happens as

```
@cuda blocks = blocksPerGrid
      threads = threadsPerBlock
      shmem = (threadsPerBlock * sizeof(Int64))
dot(a,b,c, N, threadsPerBlock, blocksPerGrid)
```

See (and execute!) the posted code.

summary and exercises

Vasily Volkov and James W. Demmel: **Benchmarking GPUs to tune dense linear algebra.** In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008. Article No. 31.

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.