Lecture 6

CUDA BASICS

Storage class qualifiers

Functions

`__global__`	Device kernels callable from host
`__device__`	Device functions (only callable from device)
`__host__`	Host functions (only callable from host)

Data

`__shared__`	Memory shared by a block of threads executing on a multiprocessor.
`__constant__`	Special memory for constants (cached)

CUDA data types

- C primitives:
  - `char`, `int`, `float`, `double`, …
- Short vectors:
  - `int2`, `int3`, `int4`, `uchar2`, `uchar4`, `float2`, `float3`, `float4`, …
  - no built-in vector math (although a utility header, `cutil_math.h`, defines some common operations)
- Special type used to represent dimensions
  - `dim3`
- Support for user-defined structures, e.g.:

```
struct particle
{
    float3 position, velocity, acceleration;
    float mass;
};
```

Library functions available to kernels

- Math library functions:
  - `sin`, `cos`, `tan`, `sqrt`, `pow`, `log`, ...
  - `sinf`, `cosf`, `tanf`, `sqrtf`, `powf`, `logf`, ...
- ISA intrinsics
  - `__sinf`, `__cosf`, `__tanf`, `__powf`, `__logf`, ...
  - `__mul24`, `__umul24`, ...
- Intrinsic versions of math functions are faster but less precise

Built-in kernel variables

`dim3 gridDim`

- number of blocks in grid

`dim3 blockDim`

- number of threads per block

`dim3 blockIdx`

- number of current block within grid

`dim3 threadIdx`

- index of current thread within block

CUDA kernels: limitations

- No variable argument lists
- No dynamic memory allocation*
  - It is possible since Kepler generation, but only when Dynamic Parallelism is enabled (it's off by default)
- No pointers-to-functions

Launching kernels

- Launchable kernels must be declared as '`__global__ void`'

```
__global__ void myKernel(paramList);
```

- Kernel calls must specify device execution environment
  - grid definition – number of blocks in grid
  - block definition – number of threads per block
  - optionally, may specify amount of allocatable shared memory per block (more on that later)

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramList);
```

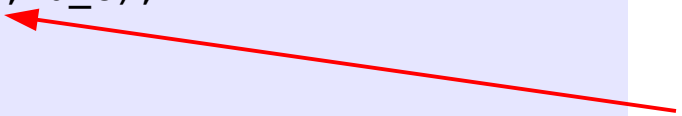
- Kernels (one block)

```
// Kernel definition
__global__ void VecAdd (float* d_A, float* d_B, float*
d_C)
{
int i = threadIdx.x;
d_C[i] = d_A[i] + d_B[i];
}

int main()
{
...

// Kernel invocation with N threads
VecAdd <<<1, N>>> (d_A, d_B, d_C);

...
}
```



Pointers to device addresses!

- Kernels (multi-block)

```
// Kernel definition
__global__ void VecAdd (float* d_A, float* d_B, float*
d_C)
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
d_C[i] = d_A[i] + d_B[i];
}

int main()
{
...

// M blocks with N threads each:
VecAdd <<<M, N>>> (d_A, d_B, d_C);
...
}
```

Thread addressing

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramlist);
```

- **`GridDef`** and **`BlockDef`** can be specified as **`dim3`** objects
  - grids can be 1D, 2D or 3D
  - blocks can be 1D, 2D or 3D

- This makes it easy to set up different memory addressing for multi-dimensional data.

Thread addressing (cont.)

- 1D addressing example: 100 blocks with 256 threads per block:

```
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```

- 2D addressing example: 10x10 blocks with 16x16 threads per block:

```
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```

- Both examples launch the same number of threads, but block and thread indexing is different
  - `kernel1` uses `blockIdx.x`, `blockDim.x` and `threadIdx.x`
  - `kernel2` uses `blockIdx.[xy]`, `blockDim.[xy]`, `threadIdx.[xy]`

Thread addressing (cont.)

- One-dimensional addressing example:

```
__global__ void kernel1(float *idata, float *odata)
{
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  odata[i] = func(idata[i]);
}
```

- Two-dimensional addressing example (dynamic arrays):

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
  int x, y, i;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  i = y * pitch + x;
  odata[i] = func(idata[i]);
}
```

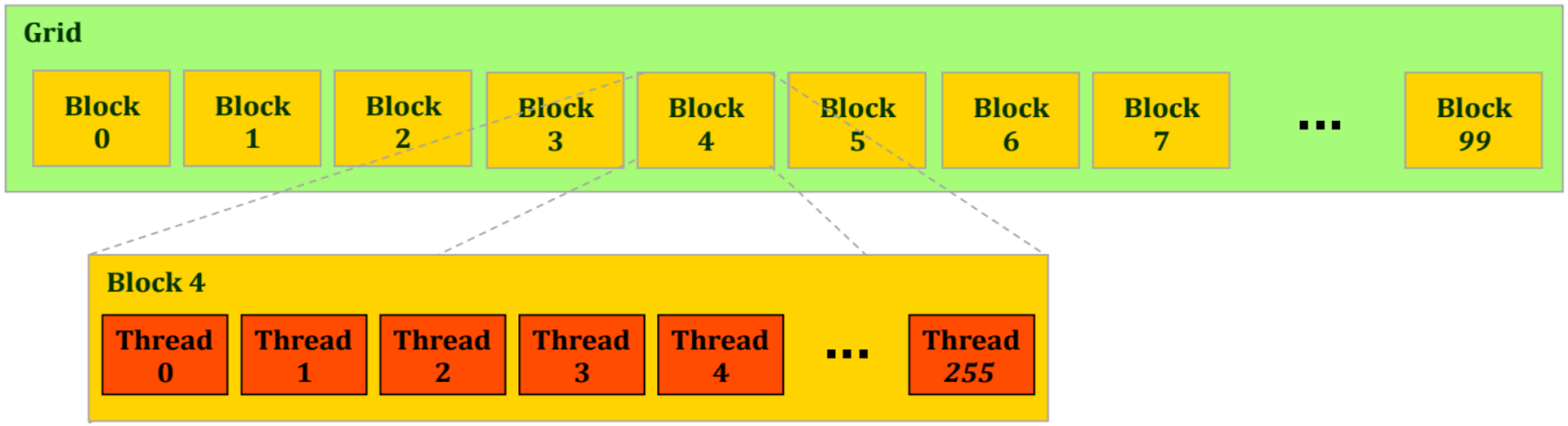
Thread addressing (cont.)

- Two-dimensional addressing example (static arrays):

```
__global__ void kernel3()
{
  int x, y, i;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  odata[y][x] = func(idata[y][x]);
}
```

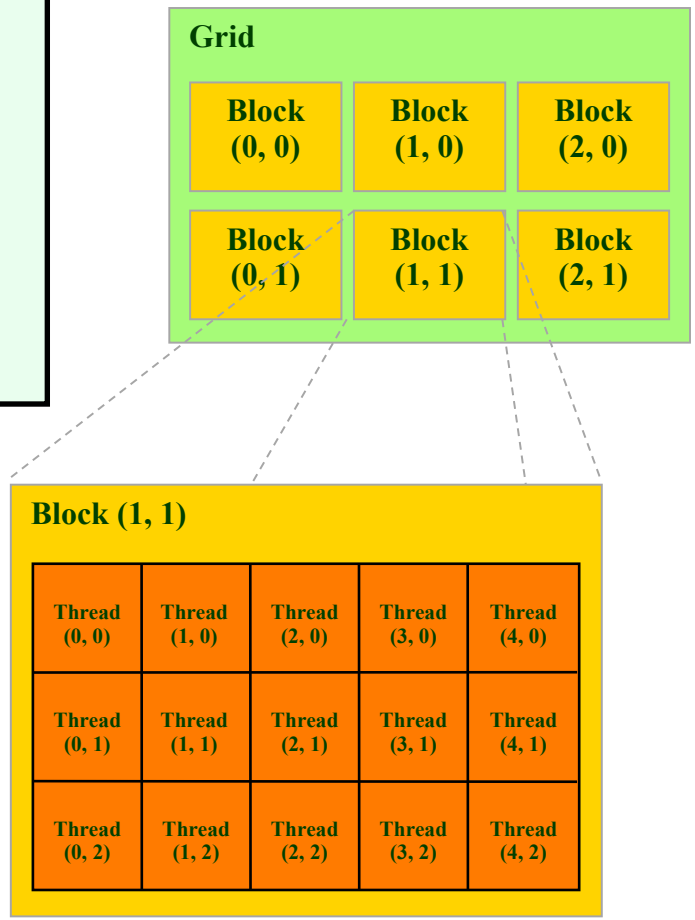
Thread addressing (cont.)

```
__global__ void kernel1(float *idata, float *odata)
{
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  odata[i] = func(idata[i]);
}
...
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```



Thread addressing (cont.)

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
  int x, y, i;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  i = y * pitch + x;
  odata[i] = func(idata[i]);
}
...
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```



Exercise: compute Julia set on GPU

You have been provided with a code that runs on the CPU. It has a .cu extension and should be compiled with nvcc.

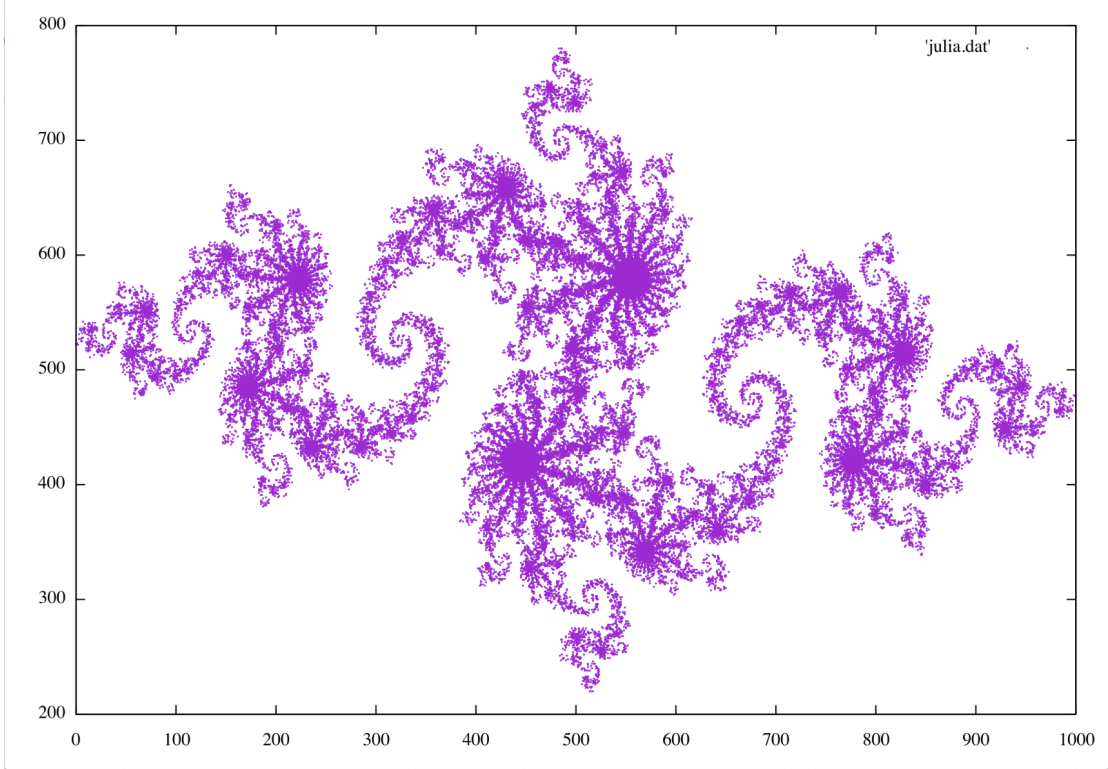
Julia set is defined as the set of points in the complex plane for which the sequence generated via repeated iterations does not diverge.

$$f(z) = z^2 + c$$

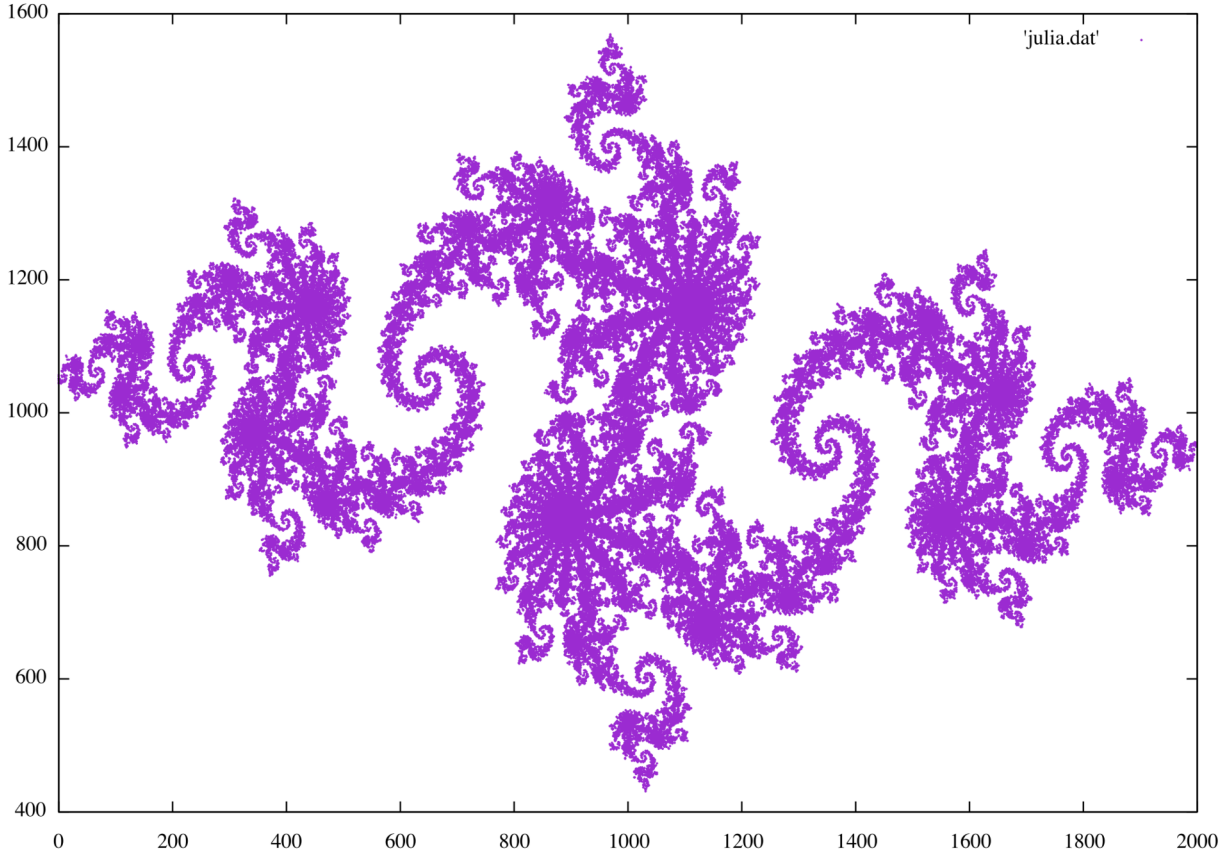
Julia set (DIM=1000, scaling=1.5)

Visualized with gnuplot:

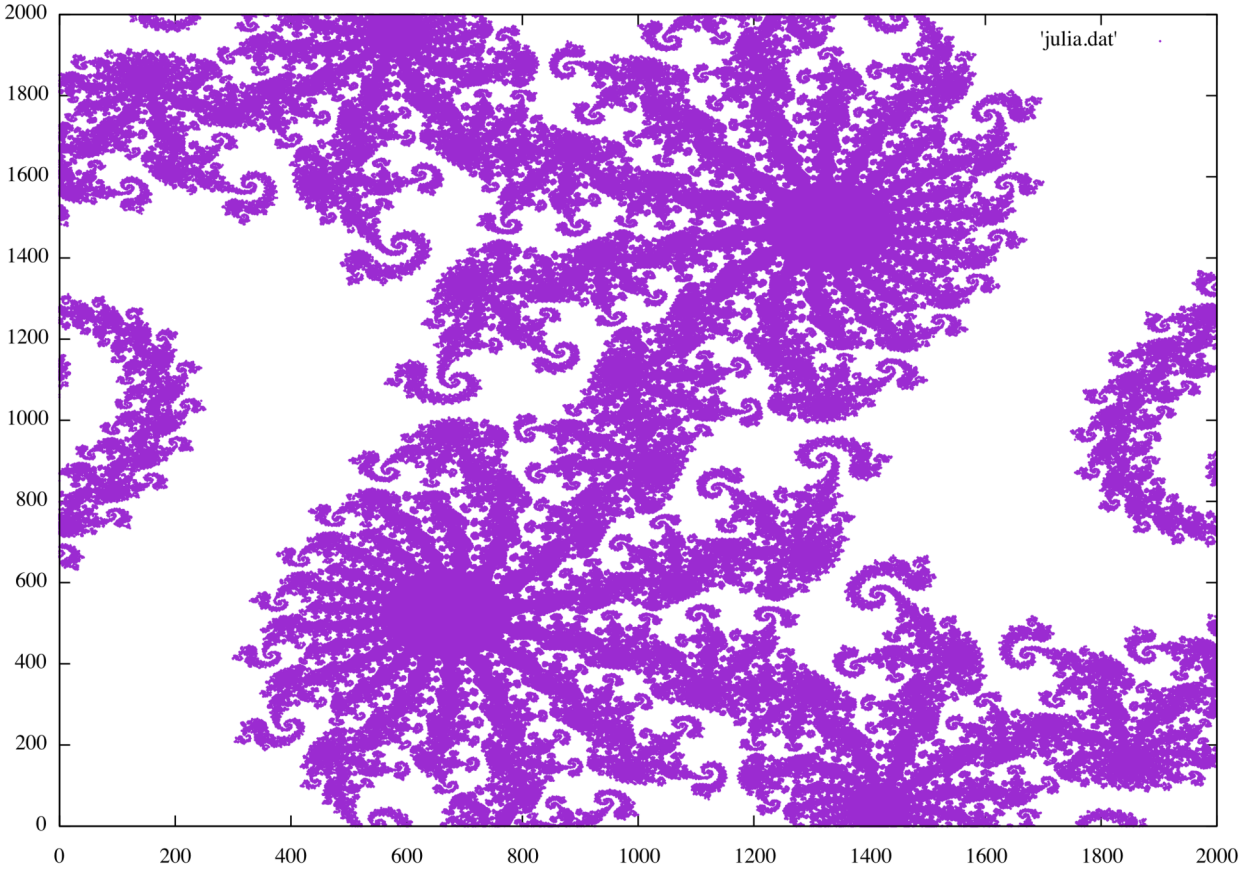
```
$ module load gnuplot
$ gnuplot
gnuplot> plot 'julia.dat' with points pointsize 0.1
```



Julia set (DIM=2000, scaling=1.5)



Julia set (DIM=2000, scaling=0.5)



Shared memory

- Much faster than global (device) memory
- Shared across the threads in a single block
- The amount is very limited, so it is often a limiting factor for CUDA optimization
- Typically statically defined, like in the following example:

```
// Device code

__shared__  double  A_loc[BLOCK_SIZE];
```

Shared memory (cont.)

- Shared memory can also be dynamically allocated, via an optional third argument inside the kernels triple angular brackets.

```
// Device code

__global__ myKernel ()
{
  extern __shared__  double  A[];
}



// Host code

MyKernel <<< N, M, SM_bytes >>> ();
```

Execution synchronization on device

- Can only be done **within a single block**.
- As a result, only up to 1024 threads can be synchronized.
- Used when a data dependence exists between different parts of a kernel.

```
// Device code

__syncthreads();
```

Execution synchronization on device (cont)

- Handling a data dependence between different parts of a kernel

```
// Kernel code
__shared__ float A[BLOCK_SIZE];

// Initializing all threads in a block:
A[threadIdx.x] = 1;

// This is needed as the previous section is executed by sequential groups of
// 32 (warp size) threads, in an undetermined order
  __syncthreads();

// A primitive example of a data dependence:
if (threadIdx.x == 0)
    for (i=0; i<BLOCK_SIZE; i++)
        sum = sum + A[i];
```

Kernel execution model

```
// Kernel code
Void MyKernel()
{

 Line1;
 Line2;
 Line3;
 …

 __syncthreads();

 Line10;
 Line11;
 Line12;
 …

 __syncthreads();

 Line20;
 Line21;
 Line22;
 …

 return;
}
```

Step 1: all warps in the block of threads execute this region sequentially, one warp at a time

Step 2: all warps in the block of threads execute this region sequentially, one warp at a time

Step 3: all warps in the block of threads execute this region sequentially, one warp at a time

Synchronization between host and device (cont)

- Kernel calls and some CUDA memory operations are executed asynchronously on host
- But in some cases you need host-device synchronization, e.g. before using a host timer, for profiling:

```
// Host code

CudaDeviceSynchronize ();
```

Synchronization between host and device (cont)

- Example: when profiling the code

```
// Host code
struct timeval  tdr0, tdr1;

gettimeofday (&tdr0, NULL);

my_kernel <<<M, N>>> ();

// Without synchronization, tdr1-tdr0 will not measure time spent inside the kernel
// (it will be much smaller):
CudaDeviceSynchronize ();
gettimeofday (&tdr1, NULL);
```

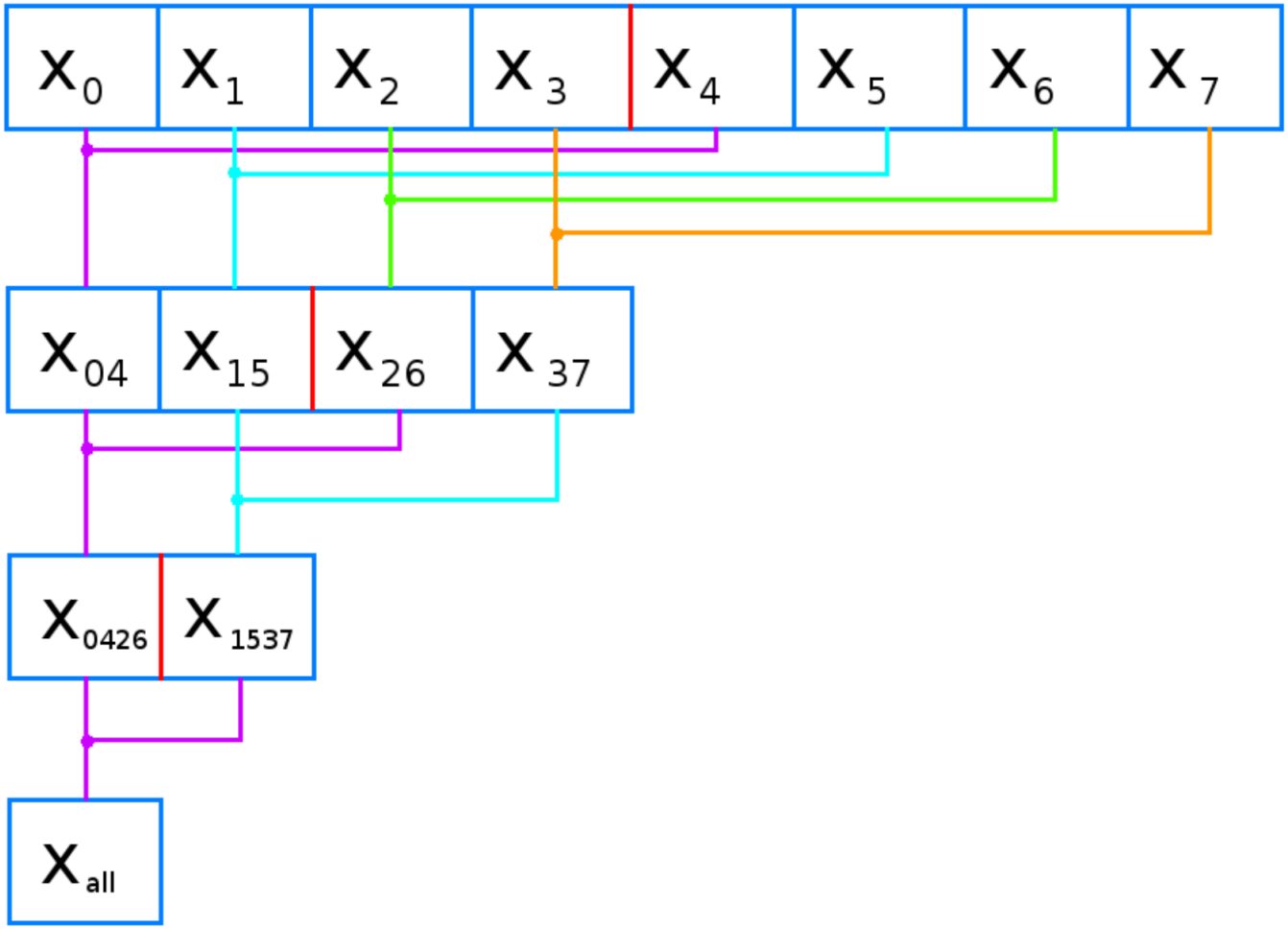
Reductions in CUDA

- Reductions: min/max, average, sum, ...
- Can be a significant bottleneck for the performance, because it breaks pure data parallelism.
- There is no perfect way to do reductions in CUDA. The two commonly used approaches (each with its own set of constraints) are
  - Binary reductions
  - Atomic reductions

Binary reductions

- The most universal type of reductions (e.g., **the only way to do floating point min or max**)
- Even when using single precision (which is faster than double precision), binary summation will be more accurate than atomic summation, because it employs more accurate **pairwise summation**.
- Usually the more efficient way to do reductions: time scales as $\log_2(N)$ **as long as there are enough of free GPU cores**.

Binary reductions (cont)



Binary reductions (cont)

- But: typically relies on (very limited) shared memory – placing constraints on how many reductions per kernel one can do
- Relies on thread synchronization, which can only be done within a single block – places constraints on how many threads can participate in a binary reduction (maximum 1024)
- For a large number of data elements (>1024), this leads to the need to do multi-level binary reductions, with storing the intermediate data in device memory; this can reduce the performance
- Can be less efficient for small number of data elements (<64)
- Significantly complicates the code

- Examples: binary summation with the number of elements being a power of two. The result is in sum[0].

```
__shared__ double  sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x;  // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1)
{
  int halfPoint = nTotalThreads / 2;    // Number of active threads

  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint; // the second element index
    sum[threadIdx.x] += sum[thread2];  // Pairwise summation
  }
  __syncthreads();
  nTotalThreads = halfPoint;  // Reducing the binary tree size by two
}
// The result is in sum[0]
```

- Examples: binary min/max with the number of elements being a power of two.

```
__shared__ double  min[BLOCK_SIZE];
...
__syncthreads(); // To make sure all min[] elements were initialized
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
  int halfPoint = nTotalThreads / 2;    // Number of active threads
  if (threadIdx.x < halfPoint)  {
    int thread2 = threadIdx.x + halfPoint; // the second element index
    double temp = min[thread2];
    if (temp < min[threadIdx.x])
       min[threadIdx.x] = temp;
    }
  __syncthreads();
  nTotalThreads = halfPoint;  // Reducing the binary tree size by two

// The result is in min[0]
}
```

- Examples: multiple binary reductions.

```
__shared__ double  min[BLOCK_SIZE], sum[BLOCK_SIZE];
...
__syncthreads(); // To make sure all array elements were initialized
int nTotalThreads = blockDim.x;

while(nTotalThreads > 1)
{
  int halfPoint = nTotalThreads / 2;    // Number of active threads
  if (threadIdx.x < halfPoint)  {
    int thread2 = threadIdx.x + halfPoint;
    sum[threadIdx.x] += sum[thread2];  // First reduction

    double temp = min[thread2];
    if (temp < min[threadIdx.x])
       min[threadIdx.x] = temp;   // Second reduction
    }
  __syncthreads();
  nTotalThreads = halfPoint;  // Reducing the binary tree size by two
}

// The result is in sum[0], min[0]
```

- Examples: two-level binary reduction

```
// Host code
#define  BSIZE  1024  // Always use a power of two; can be 32...1024
// Total number of elements to process: 1024 < Ntotal < 1024^2

int Nblocks = (Ntotal + BSIZE - 1) / BSIZE;

// Low level (the results should be stored in global device memory):
x_prereduce <<< Nblocks, BSIZE >>> ();

// High level (will read the input from global device memory):
x_reduce <<< 1, Nblocks >>> ();
```

- Examples: binary reduction with an arbitrary number of elements (BLOCK_SIZE).

```
__shared__ double  sum[BLOCK_SIZE];
 ...
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim_2; // Total number of threads, rounded up to the next power of two

while(nTotalThreads > 1)
{
  int halfPoint = nTotalThreads / 2; // Number of active threads

  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint;
    if (thread2 < blockDim.x)  // Skipping the fictitious threads blockDim.x ... blockDim_2-1
          sum[threadIdx.x] += sum[thread2];  // Pairwise summation
  }
  __syncthreads();
  nTotalThreads = halfPoint;  // Reducing the binary tree size by two
}
// The result is in sum[0]
```

- Continued: binary reduction with an arbitrary number of elements.
    - You will have to compute blockDim_2 (blockDim.x rounded up to the next power of two), either on device or on host (and then copy it to device). One could use the following function to compute blockDim_2, valid for 32-bit integers:

```
int NearestPowerOf2 (int n)
{
  if (!n) return n;  // (0 == 2^0)

  int x = 1;
  while(x < n)
    {
      x <<= 1;
    }
  return x;
}
```

Atomic reductions

- Very simple code
  - Almost no change compared to the serial code
  - A single line code: much better for code development and maintenance
  - No need for multiple intermediate kernels (saves on overheads related to multiple kernel launches)
  - Requires no code changes when dealing with any number of data elements – from 2 to millions
- Can be more efficient when the number of data elements is small (<64)

Atomic reductions (cont)

- But: atomic operations are serialized, which usually means **much** worse performance
- No atomic functionality for some basic reductions (like floating point min / max)
- A commonly employed good compromise is to use binary reduction at the lower level, and then use atomic reduction at the higher level.
- All the above means that to find the right way to carry out a reduction in CUDA, with the right balance between code readability, efficiency, and accuracy, one often has to try different approaches, and choose the most efficient.

- Examples: atomic reductions.

```
// In global device memory:
__device__  float xsum;
__device__  int isum, imax;

// In a kernel:
float x;
int i;
__shared__  imin;
...
atomicAdd (&xsum, x);
atomicAdd (&isum, i);
atomicMax (&imax, i);
atomicMin (&imin, i);
```

  - Some other atomic operations:
    - atomicExch, atomicAnd, atomicOr

- Binary at the lower level, atomic at the higher level

```
__shared__ float  sum[BLOCK_SIZE];
// Initialize sum[] array here
__syncthreads(); // To make sure all sum[] elements were initialized
int nTotalThreads = blockDim.x;  // Total number of active threads;
// only the first half of the threads will be active.

while(nTotalThreads > 1){
  int halfPoint = nTotalThreads / 2; // Number of active threads

  if (threadIdx.x < halfPoint)
  {
    int thread2 = threadIdx.x + halfPoint; // the second element index
    sum[threadIdx.x] += sum[thread2];  // Pairwise summation
  }
  __syncthreads();
  nTotalThreads = halfPoint;  // Reducing the binary tree size by two
}
if (threadIdx.x == 0)
    atomicAdd (&xsum, sum[0]); // Atomic reduction
```

Hands on exercise

- Reduction: implementing hybrid reduction scheme