Computational Physics with GPUs

Lecture 2: A first course in CUDA

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

41st Heidelberg Physics Graduate Days
Heidelberg, October 8–12, 2018

Outline

Slides and exercises

Check out the lecture notes and example code at

`http://users.complexity-coventry.org/~weigel/GPU/`

Excursion: Pointers in C

Pointers are an important element of C, contributing essentially to its flexibility and efficiency, but also leading easily to **horrible mistakes**!

A pointer is a variable storing the address in memory of another variable. Pointers are declared with a preceding $*$, the ampersand & takes a variable's address, and the asterisk $*$ dereferences the pointer.

Example:

```
int  var = 20;   // declare variable
int  *ip;        // declare pointer
char *cp;        // another pointer

ip = &var;       // store address of var in pointer
cp = &var;       // error! cp must be the address of an int

printf("Address of var variable: %p\n", &var);
printf("Address stored in ip variable: %p\n", ip);
printf("Value of *ip variable: %d\n", *ip);
```

Some valid pointers:

```
int    *ip;    // pointer to an integer
float  *fp;    // pointer to a float
float **fpp;   // pointer to a pointer
void   *vp;    // pointer to void
```

Excursion: Pointers in C (cont'd)

If a pointer does not point to a valid memory location, it is good practice to initialize it with the value NULL (which is equal to 0),

```
int *p = NULL;
p = malloc(10*sizeof(int));
if(!p) printf("error allocating memory!\n);
```

Arrays in C are stored in sequential order, so pointers can be used to inspect and manipulate arrays:

```
int var[] = {1, 2, 3};
int *p;
p = &var[0]; // point to the first element
p = var;     // an equivalent expression
++p;         // p now points to the second element
if (p > var)
  printf("p points to an element further to the right than var");
```

Strings in C are stored as arrays of characters:

```
char str[10] = "hello";
str[0]; // returns char 'h'
str;    // address of the first character of the string
```


Excursion: Pointers in C (cont'd)

Sometimes it is even useful to have a pointer to a pointer!

```
int a = 2;
int *ap = &a;
int **app = &ap;
printf("%d = %d = %d\n", a, *ap, **app);
```

This is particularly useful for two-dimensional arrays:

```
int **a;

a = (int**)malloc(10*sizeof(*int));
for(int i = 0; i < 10; ++i)
  a[i] = (int*)malloc(20*sizeof(int));

a[8][15] = 12;
```

Pointers are needed in C to alter objects in functions:

```
void f1(int a) {
  a = 10;
}
void f2(int *a) {
  *a = 10;
}

int b = 20;
f1(b); printf("%d\n", b);
f2(&b); printf("%d\n", b);
```


GPU workflow

The basic workflow for a GPU code proceeds along the following lines:

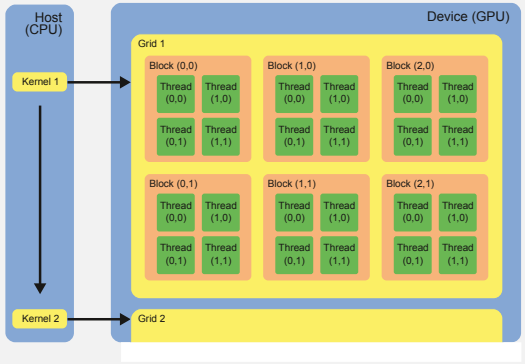
- Initialize input data (matrices, particle configurations, ...) on CPU.
- Allocate memory on GPU and copy data from CPU to GPU.
- Execute kernel(s) on GPU, processing the data.
- Copy the results back from GPU to CPU.
- Iterate or exit.

More elaborate variations:

- Use of unified virtual addressing.
- Part or full off-loading of calculations to GPU.
- Interleaving computation and communication.
- Integration with MPI.

Kernel execution

Kernel	GPU program that runs on a grid of threads
Thread	scalar execution unit
Warp	block of 32 threads executed in lockstep
Block	a set of warps executed on the same SM
Grid	a set of blocks usually executed on different SMs



Execution configuration

Function qualifiers

```
__global__ void f()
```

- function called from host, executed on device
- must return void

```
__device__ int f()
```

- function called from device, executed on device

```
__host__ int f()
```

- function called from host, executed on host
- `__host__` and `__device__` can be combined to generate CPU and GPU code

Built-in variables

All `__global__` and `__device__` functions have the following automatic variables:

- `dim3 gridDim;` — dimension of the grid in blocks
- `dim3 blockDim;` — dimension of the block in threads
- `dim3 blockIdx;` — block index within grid
- `dim3 threadIdx;` — thread index within block

The indices can be used to construct a global thread index, for instance for a block size of 5 threads,

```
thread_index = blockIdx.x*blockDim.
    x + threadIdx.x;
```


Execution configuration

Execution configuration

Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...);
```

Execution configuration:

- `dG`: dimension and size of grid in blocks
  - two-dimensional, `dG.x` and `dG.y`
  - total number of blocks launched is `dG.x` × `dG.y`
- `dB`: dimension and size of each block
  - two- or three-dimensional, `dB.x`, `dB.y`, and `dB.z`
  - total number of threads per block is `dB.x` × `dB.y` × `dB.z`
  - if not specified, `dB.z = 1` is assumed

Built-in variables

All `__global__` and `__device__` functions have the following automatic variables:

- `dim3 gridDim;` — dimension of the grid in blocks
- `dim3 blockDim;` — dimension of the block in threads
- `dim3 blockIdx;` — block index within grid
- `dim3 threadIdx;` — thread index within block

The indices can be used to construct a global thread index, for instance for a block size of 5 threads,

```
thread_index = blockIdx.x*blockDim.
    x + threadIdx.x;
```


Quiz

Possible grid dimensions are specified in the programming guide,

`https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities`

Index 1

If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to the data index?

(a) `i=threadIdx.x+threadIdx.y`

(b) `i=blockIdx.x+threadIdx.x`

(c) `i=blockIdx.x*blockDim.x+threadIdx.x`

(d) `i=blockIdx.x*threadIdx.x`

Index 2

We want to use each thread to calculate two adjacent elements of a vector addition. What mapping is correct if `i` is the index of the first element?

(a) `i=blockIdx.x*blockDim.x+threadIdx.x+2`

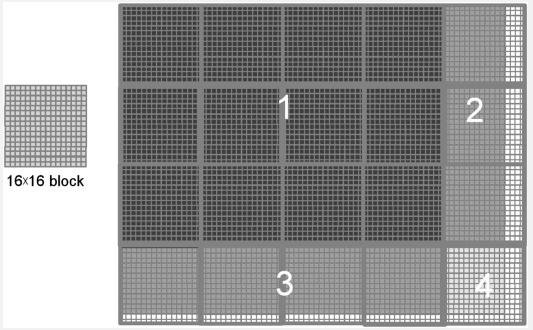
(b) `i=blockIdx.x*threadIdx.x*2`

(d) `i=blockIdx.x*blockDim.x*2+threadIdx.x`

Thread mapping

The organization of threads in a (1D or 2D) grid of (1D, 2D or 3D) blocks is usually chosen to match the dimensionality and structure of the input data.

For example, a 2D grid of 2D blocks would probably be used to work on the pixels of an image.

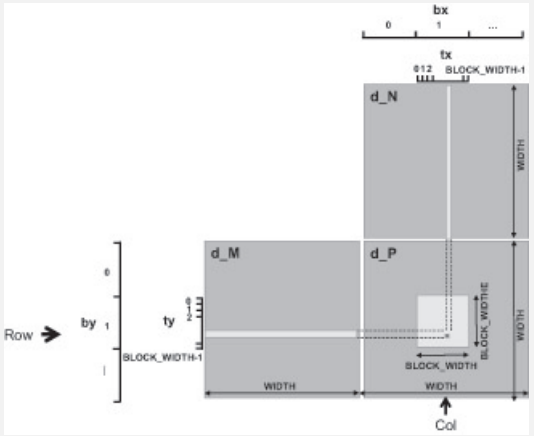


Note that (dynamic) arrays in C need to be *linearized*: $M_{ij} = \mathtt{M[i*n+j]}$ (row-major).

Example: matrix multiplication

Consider multiplication of two square, `Width` × `Width` matrices `d_M` and `d_N`, `d_P` = `d_M` · `d_N`.

How should one map threads to data elements? One option is to use **one thread per output element** $\texttt{d\_P}_{i,j}$.



Matrix multiplication: a simple kernel

Matrix multiplication kernel

```
__global__ void MatrixMult(float* d_M, float* d_N, float* d_P, int Width) {
  // Calculate the row index of the d_P element and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }
}
```


Matrix multiplication: a simple kernel (cont'd)

In the host code, the split of the matrix into blocks needs to be organized, probably using an adjustable block size

(Part of) the host code

```
#define BLOCK_WIDTH 16

int NumBlocks = Width/BLOCK_WIDTH;
if(Width % BLOCK_WIDTH) NumBlocks++;
dim3 dimGrid(NumBlocks, NumBlocks);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

MatrixMult<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

A few notes are in order:

- we need to ensure that we have enough threads to cover all matrix elements
- the excess threads are then "masked away" in the kernel code
- one might want to optimize over difference choices of `BLOCK_WIDTH` for best performance

Thread scheduling and synchronization

In CUDA, all threads of a grid run the same code, but we can take branches by using if conditions based on `threadIdx.x` and `blockIdx.x` etc.

Threads in the same **block** can by synchronized using `__syncthreads()`: all threads of a block halt there until the last thread has reached this point.

(Note that hence there will be a deadlock if threads in the same block can take different branches containing `__syncthreads()` statements!)

On the other hand, threads in **different** blocks are independent and cannot be synchronized!

(Unless with tricks via the use of global memory.)

These principles lead to good results for devices with very different scopes of available resources, known as transparent scalability.

To understand the effect of scheduling, one needs to take into account the resource limits of the multiprocessors, which can be checked at

`https://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities`

(And they can be queried using `cudaGetDeviceProperties()`.)

Quiz

Resident threads

If a CUDA device's SM can take up to 1536 threads and up to 4 thread blocks, which of the following block configurations would result in the most number of threads in the SM?

(a) 128 threads per block

(b) 256 threads per block

(c) 512 threads per block

(d) 1024 threads per block

Threads in the grid

For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

(a) 2,000

(b) 2,024

(c) 2,048

(d) 2,096

Scheduling on CUDA devices works in warps of 32 threads that operate in lockstep, always operating exactly the same code. If a conditional evaluates differently for some threads in a warp, the code needs to run multiple times, once for each outcome. This is called

Memory hierarchy

Per thread

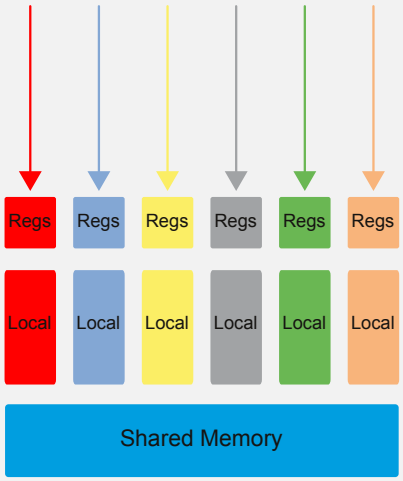
- Registers (extra fast, no copy for ops)
- Local memory

Thread blocks: shared memory

- allocated by thread block, same lifetime as block
- allocate as

```
__shared__ int s_array[DIM];
```

- low latency (of the order of 10 cycles), bandwidth up to 1 TB/s
- use for data sharing and user-managed cache



Memory and performance

Memory accesses are most often the performance limiting factor in GPU applications.

In the first matrix multiplication kernel, the main loop is

```
for (int k = 0; k < Width; ++k) {
  Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
```

which loads two values, stores one value and performs a multiplication and an addition. Hence the *compute to global memory access ratio* is 1.

Imagine we are using a card which has a memory bandwidth of about 200 GB/s and a peak single precision FP performance of 1500 GFlop/s.

With 4 bytes per float, the maximum performance of this kernel is $200/4 = 50$ GFlop/s, which is just 3% of the peak performance! To arrive at the peak performance, we would need 30 FP operations per global memory access.

Memory hierarchy

Per device: global memory

- accessible to all threads on device
- lifetime is user-defined

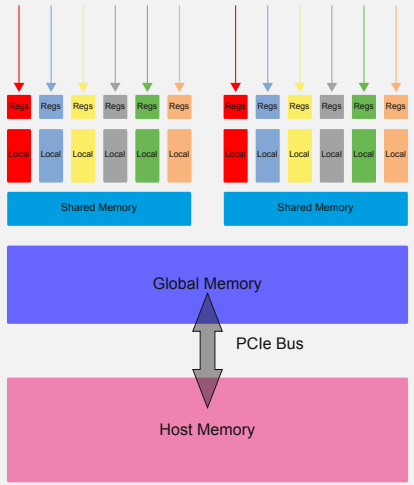
```
cuda_malloc(void **pointer,
     size_t nbytes);
cuda_free(void* pointer);
```

- latency several hundred clock cycles
- bandwidth ≈ 160 GB/s on Fermi (access pattern needs to conform to **coalescence rules** for good performance)

Per host: device memory

- no direct access from CUDA threads
- copy data to/from device with

```
cudaMemcpy(void* dest, void*
     src, size_t nbytes,
     cudaMemcpyHostToDevice);
```

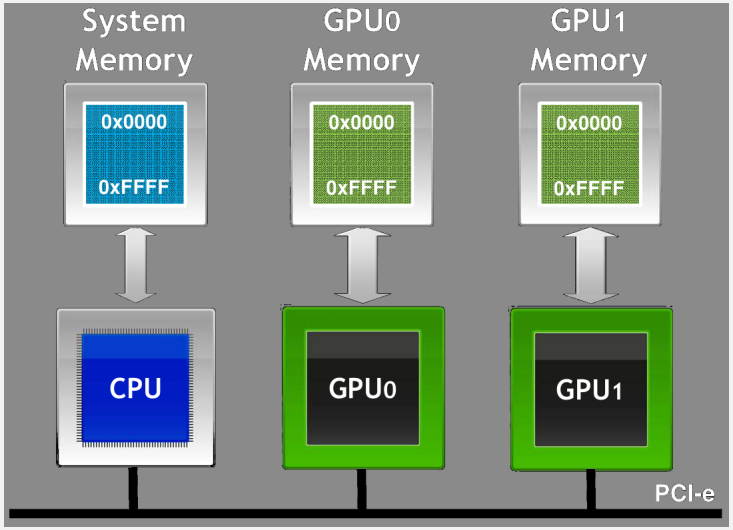


Memory hierarchy (summary)

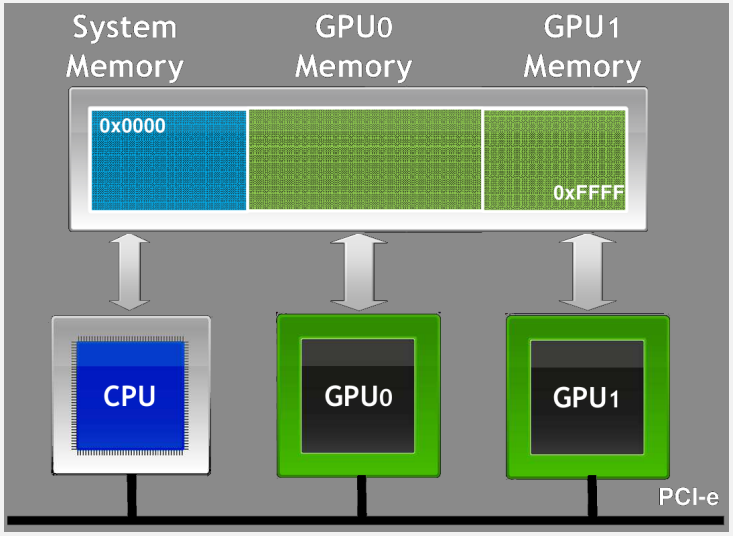
More generally, the different types of memory have the following characteristics:

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	(Yes)	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Unified virtual addressing



Unified virtual addressing



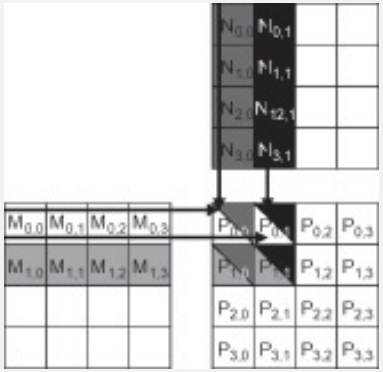
CUDA variables

Variable declaration	Memory	Scope	Lifetime	Penalty/Latency
`int var;`	register	thread	thread	1X
`int array_var[10];`	local	thread	thread	100X (pre-Fermi)
`__shared__ int shared_var;`	shared	block	block	10X
`__device__ int global_var;`	global	grid	application	100X
`__constant__ int constant_var;`	constant	grid	application	1X

- automatic scalar variables reside in registers, compiler will spill into local memory in shortage of registers
- automatic array variables (in the absence of qualifiers) reside in thread-local memory
- the type of memory used will be crucial for the performance of the application

Maxtrix multiplication (again)

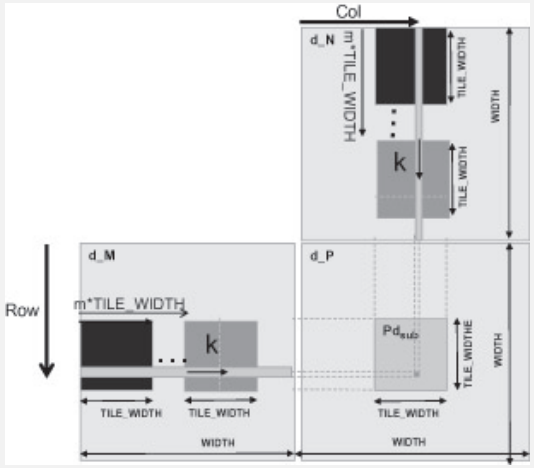
In the way we have set up the multiplication kernel,



each value will be loaded `Width` times from global memory!

Maxtrix multiplication (again)

We can improve things by avoiding global memory accesses via the use of shared memory. If we load tiles of both matrices into shared memory, the loaded values can be re-used.



Maxtrix multiplication (again)

We need `Width/TILE_WIDTH` tiles to to cover each row and column for the calculation of each element of the result matrix.

The reduction in global memory accesses is then a factor of `TILE_WIDTH`!

The tile width is mostly limited by the shared memory size: we need $4 \times 2 \times$ `TILE_WIDTH` bytes, e.g., a width of 64 results in 32K of shared memory used.

Maxtrix multiplication (again)

Matrix multiplication kernel

```
__global__ void MatrixMult(float* d_M, float* d_N, float* d_P, int Width) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x, by = blockIdx.y;
  int tx = threadIdx.x, ty = threadIdx.y;

  // Identify the row and column of the d_P element
  int Row = by*TILE_WIDTH+ty, Col = bx*TILE_WIDTH+tx;

  float Pvalue = 0;
  // Loop over d_M and d_N tiles
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of tile
    Mds[ty][tx] = d_M[Row*Width+m*TILE_WIDTH+tx];
    Nds[ty][tx] = d_N[(m*TILE_WIDTH+ty)*Width+Col];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
  }
  d_P[Row*Width+Col] = Pvalue;
}
```

The Julia set

Definition

Let $f(z) = p(z)/q(z)$ be a complex function, where $p(z)$ and $q(z)$ are complex polynomials.

The Julia set of f can be described as the set of points for which

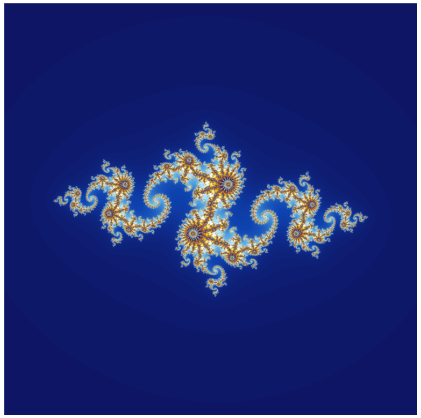
$$\lim_{n\to\infty} |f^{(n)}(z)| < \infty,$$

where $f^{(n)}(z)$ denotes the n-fold repeated application of f on z.

In general, the Julia set is a self-similar fractal. Standard example:

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

where c is a complex constant.



(Color codes are for different rates of divergence.)

Julia: CPU code 1

Driver

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

Kernel

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
 }
```

- `CPUbitmap` externally defined
- `display_and_exit()` externally defined
- `julia()` will return 1 if point in set
- choose red and black colors, respectively

Julia: CPU code 2

Julia set function

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

- shift center of image to $(0, 0)$
- re-scale to unit square in complex plane
- use `scale` factor to zoom
- arbitrary constant c in $z_{n+1} = z_n^2 + c$

Julia: CPU code 3

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

- use `cuComplex` structure to represent complex numbers
- operator overloading for natural semantics

Julia: GPU code 1

Driver

```
int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
    data.dev_bitmap = dev_bitmap;

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    bitmap.display_and_exit();
}
```

- use `cudaMalloc` to allocate memory on device
- assign one thread per pixel, one thread per block, resulting in a `DIM` $\times$ `DIM` grid
- third dimension in `dim3` defaults to one $\Rightarrow$ 2D grid

Julia: GPU code 2

Kernel

```
__global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

- `__global__` qualifier for device function to be called from host
- each thread gets `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, and `blockIdx.y`
- translate to `offset` in image array
- note that `for` loops have gone away
- four chars to represent R, G, B and alpha channels

Julia: GPU code 3

Julia function

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

- `__device__` qualifier for device function to be called from device code
- identical to CPU code apart from function qualifier

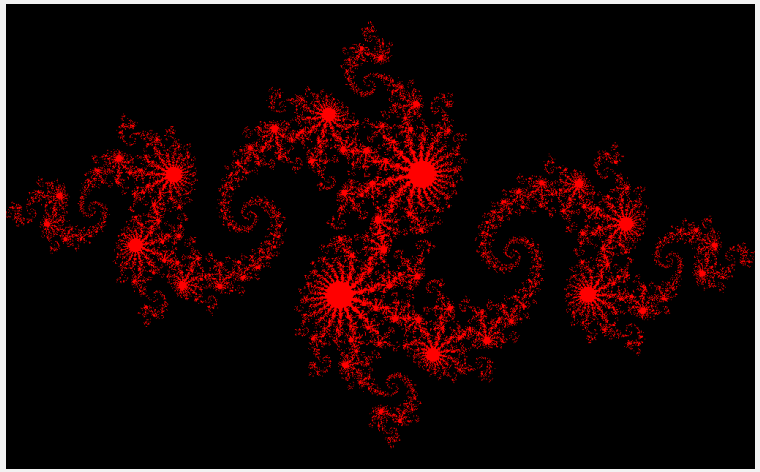
Julia: GPU code 4

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    __device__ cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

- almost identical to CPU version
- only difference are `__device__` function qualifiers

Julia: result



Play around with code, cf.

J. Sanders, E. Kandrot: "*CUDA by example — An Introduction to General-Purpose GPU Programming*", (Addison Wesley, Upper Saddle River, 2011).

Julia: GPU code — improvements

- On Fermi, maximum number of resident blocks per SM is 8 (16 for Kepler), hence 24 out of 32 cores (176 out of 192 for Kepler) are idle
- To improve, one could introduce tiles of, say, 16×16 pixels. How would the program need to be modified?
- could have used virtual unified addressing to eliminate the CPU copy of the image
- could integrate with OpenGL for interactive rendering
- ...

Summary and outlook

This lecture

You should by now by quite comfortable with the basic ideas of CUDA programming and have some feeling for what is important to get reasonable performance.

Next lecture

In lecture 3, we will apply these ideas to the problem of simulating classical spin models with local update algorithm. Suitable tuning will result in several 100-fold speed-ups.

Reading

If using the book by Kirk and Hwu, you could work through all material until Chapter 5. More about the Julia set problem on GPU can be found in Sanders and Kandroot: *CUDA by example*.