Recitation 5

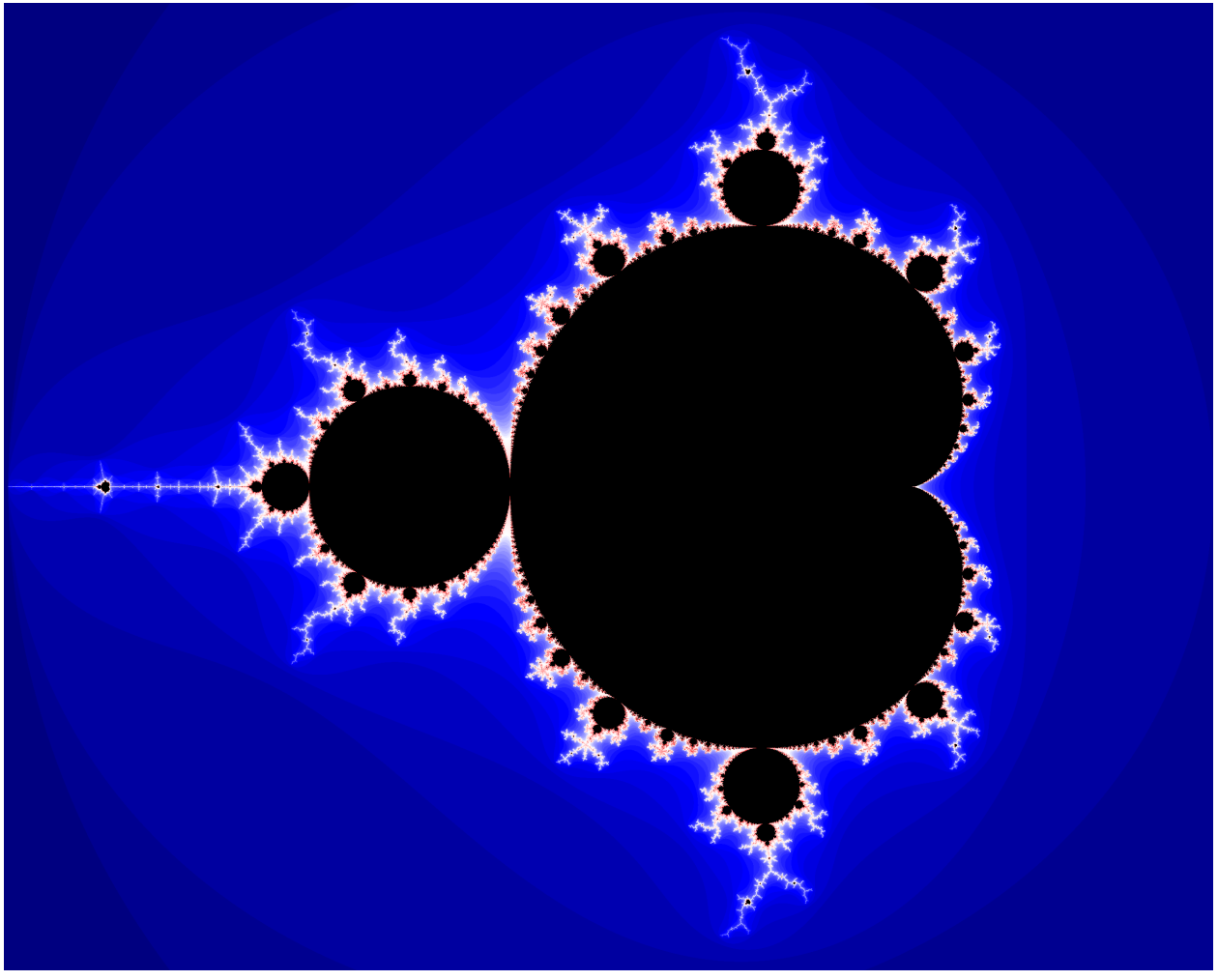
Announcements

- Assignment 5, theory + code, pass/fail, deadline 02.11

GPUs

- For this assignment and the next you'll need a programable NVIDIA GPU.
- tdt4200s1.idi.ntnu.no do not have GPUs, and cannot be used.
- The PCs in 015 IT-south can be used (also remotely, its-015-xx.idi.ntnu.no)
- If you have a recent NVIDIA card, you can do the problem set on your own machine.
- BUT, different cards, and different versions of the CUDA framework are not always compatible. You should always make sure your solution works on the seutp we have in the lab.

Mandelbrot



GPU programing model

- Most of the program runs on the CPU, but certain computationaly heavy functions are offloaded to the GPU.
- These functions are known as *kernels*
- Before you execute the kernels, you usually have to manually move input data to the GPU, and when the kernels finish, you have to manually move it back.
- While the kernel is executing, the CPU is usually just waiting.
- The CPU is ofte called the *host*, and the GPU the *device*.

Programing model

- When you call or launch a kernel, you start many threads, all running the same code.
- But each thread can find its thread ID, and branch based on it, so even if all the threads execute the same program, they can take different paths through the code.
- This model is known as SPMD, Single Program, Multiple Data.

SPMD

```
void mult_or_add(float* a, float* b, float*c){
  int threadId = blockIdx.x * blockDim.x +
                 threadIdx.x;
  if(threadId > 4){
    c[threadId] = a[threadId]*b[threadId];
  }
  else{
    c[threadId] = a[threadId]+b[threadId];
  }
}
```

Memory allocation

CUDA

```
float* a_device;
cudaMalloc((void**)&a, sizeof(float)*1024);
```

- The CPU and GPU typically have (physically) separate memories.
- The host is responsible for allocating memory, and transfering input (if needed) to the GPU.
- Allocates memory on the device/GPU, (like malloc on the host).

Memory transfer

CUDA

```
cudaMemcpy(device_array, host_array,
  1024*sizeof(float),
  cudaMemcpyHostToDevice);
cudaMemcpy(host_array, device_array,
  1024*sizeof(float),
  cudaMemcpyDeviceToHost);
```

- Used to transfer data from host to device memory, and back.

Launching kernels, CUDA

CUDA

```
my_kernel<<<nBlocks, nThreads>>>(arg1, arg2);
```

- Launces the kernel on the GPU, in parallel.
- `nBlocks` blocks, each consisting og `nThreads` threads are launced. (Total number of threads is `nBlocks` * `nThreads`)

Launching kernels

- When we call a kernel, we start several threads, which all execute the same code.
- The threads are organized in two levels: threads are organized into blocks, and block into grids.
- The total number of threads is often fixed by the problem. But the best blocksize is a not allways easily determined (we'll come back to this later).
- Both blocks and grids can be one, two or three dimensional, so both `numBlocks` and `threadsPerBlock` can be vectors.

Finding thread ID

CUDA

```
int id = blockIdx.x * blockDim.x + threadIdx.x;
```

- Functions/built in variables to find global/local id as well as local and global sizes.
- These examples are for 1D grids.

Finding thread identity

- `blockIdx` and `threadIdx` are built in variables storing the ID of your block, and thread id within that block.
- That is, the `threadIdx` is not the global thread ID.
- Since both blocks and grids can be one, two, or three dimensional, both `threadIdx` and `blockIdx` are three component vectors, so we have to use their members, i.e. `threadIdx.x` or `blockIdx.y`.

2D example

CUDA

```
dim3 gridBlock, threadBlock;
gridBlock.y = 4; gridBlock.y = 4;
threadBlock.x = 5; threadBlock.x = 5;
my_kernel<<<gridBlock, threadBlock>>>(...)

__global__ my_kernel(...){
    int x = blockIdx.x *
        blockDim.x + threadIdx.x;
    int y = blockIdx.y *
        blockDim.y + threadIdx.y;
    ...
}
```

Code organization, CUDA

- For small programs, all the code is put into a .cu file.
- Functions can be decalred with three different qualifiers:
- `__global__` Device code that can be called from the host.
- `__device__` Device code that can only be called from the device.
- `__host__` Host code. Is the deafault, and can be omited.
- You can add both `__host__` and `__device__` for functions which shold be available on both the host and device.

Asynchronous

- Kernel launches are asynchronous. The kernel launch returns immediately on the host, before the GPU is finished.
- You can wait for the GPU to finish with the `cudaDeviceSynchronize()` function.
- Copying memory back to the host will do this implicitly.
- If you launch multiple kernels, one kernel will not start until the previous has finished.

Error checking

- Most/all cuda functions on the host return a `cudaError_t`
- It can be useful to check this, because it gives errors if you call functions incorrectly etc.
- You can translate it into a string with `cudaGetErrorString()`.
- Another possibility is to place: `printf("%s"), cudaGetErrorString(cudaGetLastError()))` at strategic locations.
- The infamous ”unspecified launch failure” is typically the equivalent of a segfalut on the CPU.

Debugging

- It is possible (with compute capability 2.0) to put `printf()`s in the kernel code.
- You should make sure just one thread prints to avoid to much output.

Etc.

- If your GPU is connected to a display, there is a watchdog timer that will kill any kernel after approx. 10 seconds.
- So if your programs suddenly stops working for large problems sets, this might be the problem.

Mandelbrot on the GPU

- The computation for each pixel /point in the complex plane is independent. We can therefore lauch one GPU thread for each pixel.
- You will need to complete two functions, one which is the actuall kernel, doing the mandelbrot computations on the GPU, and one (CPU) function to allocate memory, launch the kernel, and transfer the result back.
- The provided code also includes the CPU computation (for time comparison).

Compilation

CUDA

```
$ nvcc -o test test.cu
$ ./a.out
```

- Cuda must be compiled with Nvidias `nvcc`.
- On the ITS 015 machines, you must also use the flag `-ccbin=g++-4.8` to use a supported version of gcc, this is currently in the Makefile, if you use a different machine, you may have to remove this.

Example program