Programming the GPU using OpenCL

Introductory Tutorial

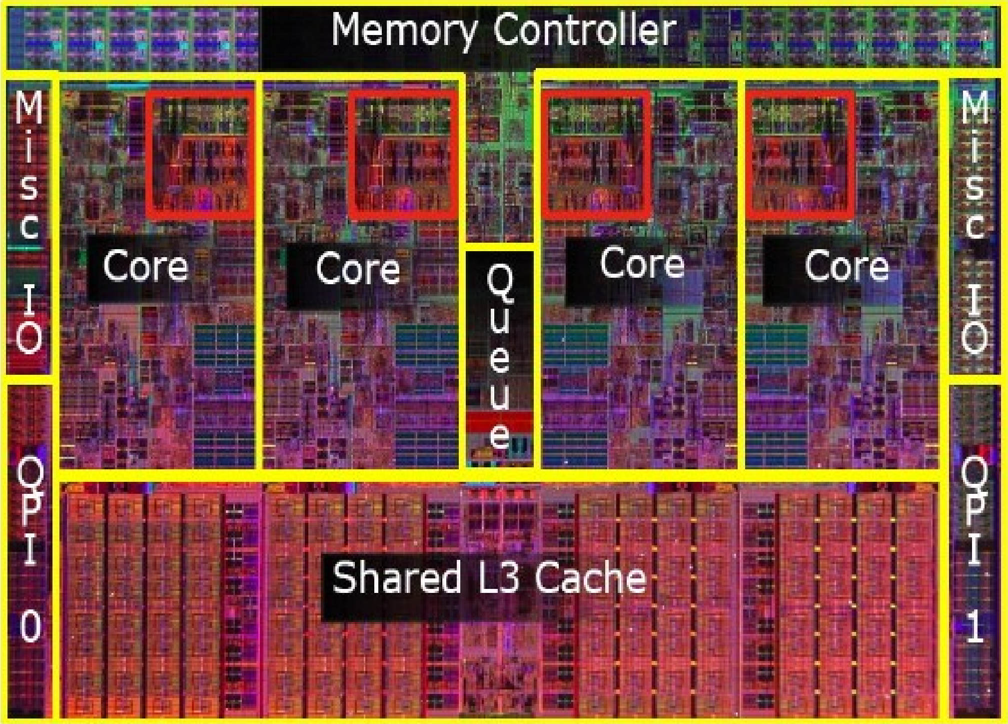
Nikolas Ladas

17/6/2010

Introduction

- CPUs are optimized for single thread performance
- Out of order execution, branch prediction and large caches take up most of the chip's area
- These features are not needed for data driven applications(e.g. Scientific applications, HPC)
  - predictable access patterns
  - Few control instructions
- Need more computation resources

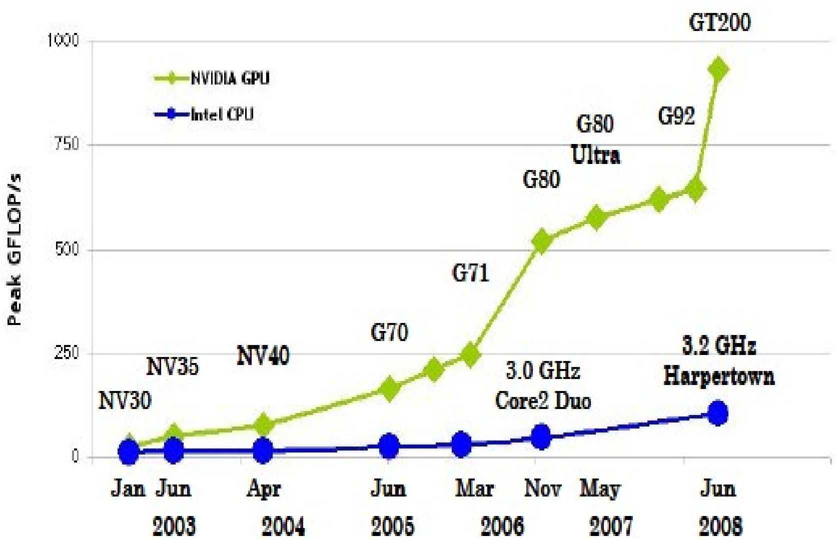
Intel i7 processor



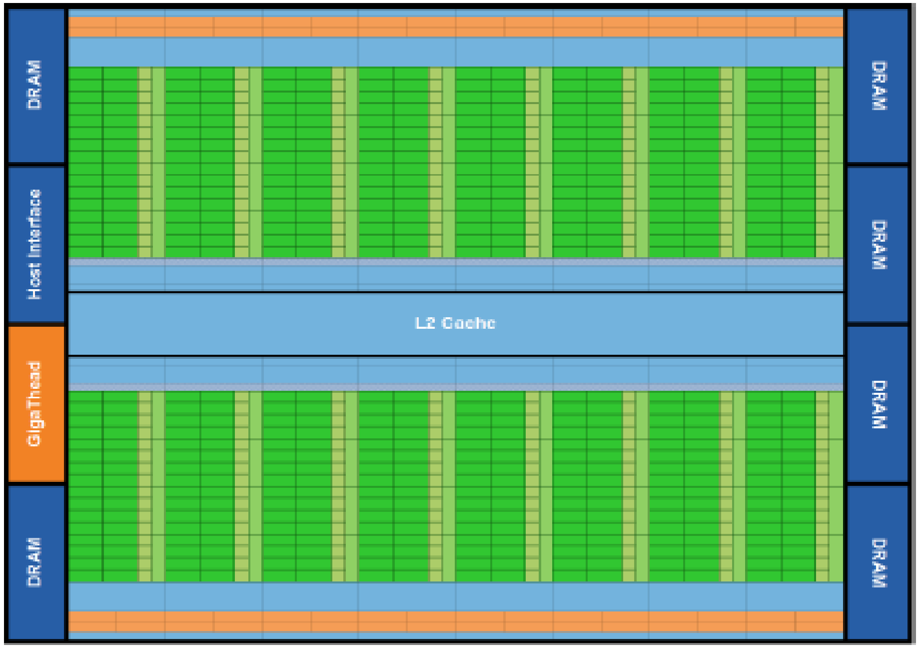
Source: NVIDIA’s Fermi: The First Complete GPU Computing Architecture, Peter N. Glaskowsky,

GPUs for Computation

- Many simple cores
- Fermi:16 stream processors*32 cores each
- On board DRAM
  - Faster to access
- → Many GFLOP/s!



NVIDIA Fermi GPU architecture



Source: www.nvidia.com

OpenCL

- Open standard for parallel programming on heterogeneous systems
  - CPU, GPU, other accelerators
  - Easy to use: C code + APIs
  - Portable: compiles automatically to the platform available
- We will focus on GPU programming

OpenCL Program Structure

- 'host' code:
  - C/C++ code that will run on the CPU - Compiled using standard compilers + OpenCL headers
  - Uses OpenCL APIs to:
    - Move data from system memory to GPU DRAM
    - Start multiple instances of the kernel to run on the GPU
    - Each instance acts on a portion of the data
    - Copy back results
- 'kernel' code
  - C code that will run on the GPU – Compiled using the vendor's compiler(e.g. NVIDIA's compiler)
  - Operates on data stored in the GPU DRAM
  - Writes results in GPU DRAM

Example: Vector Addition(Host Code)

- Query the system for available devices

```
clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
```

Example: Vector Addition(Host Code)

- Query the system for available devices

```
clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
```

- Select which device to use

```
cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
```

Use the 1st GPU available

Example: Vector Addition(Host Code)

- Query the system for available devices

```
clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
```

- Select which device to use

```
cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
```

Use the 1st GPU available

- Create the vectors

```
float * pA = new float[4096]; float * pB = new float[4096];float * pC = new float[4096];
randomize(pA);    randomize(pB);
```

Example: Vector Addition(Host Code)

- Query the system for available devices

```
clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
```

- Select which device to use

```
cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
```

Use the 1st GPU available

- Create the vectors

```
float * pA = new float[4096]; float * pB = new float[4096];float * pC = new float[4096];
randomize(pA);    randomize(pB);
```

- Compile the kernel

```
char * csProgramSource = oclLoadProgSource(VectorAdd.cl, "", KernelLength);
hProgram = clCreateProgramWithSource(hContext, 1,(const char **)&csProgramSource,
&szKernelLength, &ciErr1);
hKernel = clCreateKernel(hProgram, "VectorAdd", &ciErr1);
```

Which kernel function to use as main()

- Allocate GPU memory for the vectors

hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | **CL_MEM_COPY_HOST_PTR**, 4048 * sizeof(cl_float), **pA**, 0);

hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | **CL_MEM_COPY_HOST_PTR**, 4048 * sizeof(cl_float), **pB**, 0);

hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY , 4048 * sizeof(cl_float), 0 , 0);

- Allocate GPU memory for the vectors

```
hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float), pA, 0);

hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float), pB, 0);

hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY , 4048 * sizeof(cl_float), 0 , 0);
```

- Specify the kernel parameters

```
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);

clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);

clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
```

- Allocate GPU memory for the vectors

```
hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float), pA, 0);

hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float), pB, 0);

hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY , 4048 * sizeof(cl_float), 0 , 0);
```

- Specify the kernel parameters

```
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);

clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);

clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
```

- Run 4096 kernels (1 for each vector element)

```
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, 4096, 0, 0, 0, 0);
```

- Allocate GPU memory for the vectors

```
hDeviceMemA = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float),   pA,   0);

hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, 4048 * sizeof(cl_float),   pB,   0);

hDeviceMemC = clCreateBuffer(hContext,     CL_MEM_WRITE_ONLY ,   4048 * sizeof(cl_float), 0 , 0);
```

- Specify the kernel parameters

```
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);

clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);

clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
```

- Run 4096 kernels (1 for each vector element)

```
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, 4096,  0, 0, 0, 0);
```

- Copy results from GPU back to host memory

```
clEnqueueReadBuffer(hCmdQueue, hDeviceMemC, CL_TRUE, 0,4096 * sizeof(cl_float), pC, 0, 0, 0);
```

Blocks CPU execution until all kernels finish

Vector Addition: Kernel Code

```
//VectorAdd.cl
__kernel void VectorAdd(__global const float* a, __global const float* b, __global float* c)
{
    // get index into global data array
    int iGID = get_global_id(0);
    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}
```

Vector Addition: Kernel Code

```
//VectorAdd.cl
__kernel void VectorAdd(__global const float* a, __global const float* b, __global float* c)
{
    // get index into global data array
    int iGID = get_global_id(0);
    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}
```

Will return 0-4095

What is Allowed in the Kernel Code

- C99 code
  - No recursion
  - No function pointers
  - No standard headers
- Built-in data types
  - Scalar: char, int, float, bool...
  - Vector types: char2, char4, float16, int8...
- Vector operations

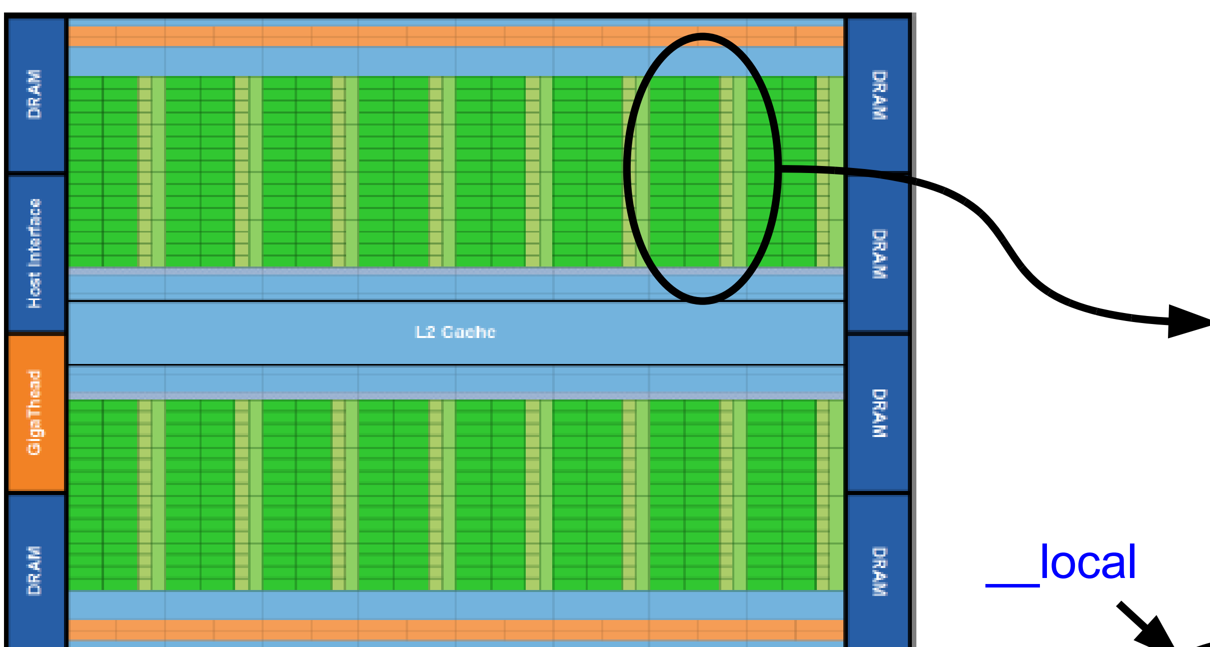
```
int4 vi1 = (int4)(0, 1, 2, 3);
vi1 = vi1 - 2
//vi1(-2, -1, 0, 1)
```

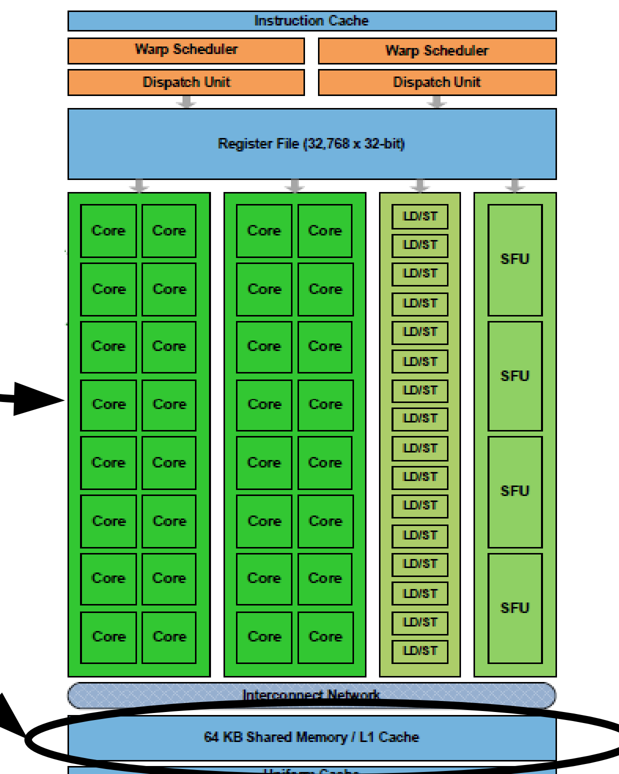
- Built-in math functions
- Synchronization primitives

Local VS Global Memory

__kernel void myKernel(__global float* A, __local float *B) {....

- __global memory is stored in GPU DRAM and cached in the L2 cache(available in Fermi GPUs only) → slow!
- __local memory shared by the cores of each stream processor → fast but limited





Local VS Global Memory cnt.

- Global memory is not coherent
  - Programmer's job to ensure coherency
- Local memory can be coherent
  - Use atomic read/write primitives(OpenCL specific)

Good Practices

- Load data from global memory to local
- Operate as much as possible on local data
- Minimize control instructions
  - Instruction issue is shared among all cores in a stream processor
  - If control flow diverges threads are serialized

Not Covered in This Tutorial

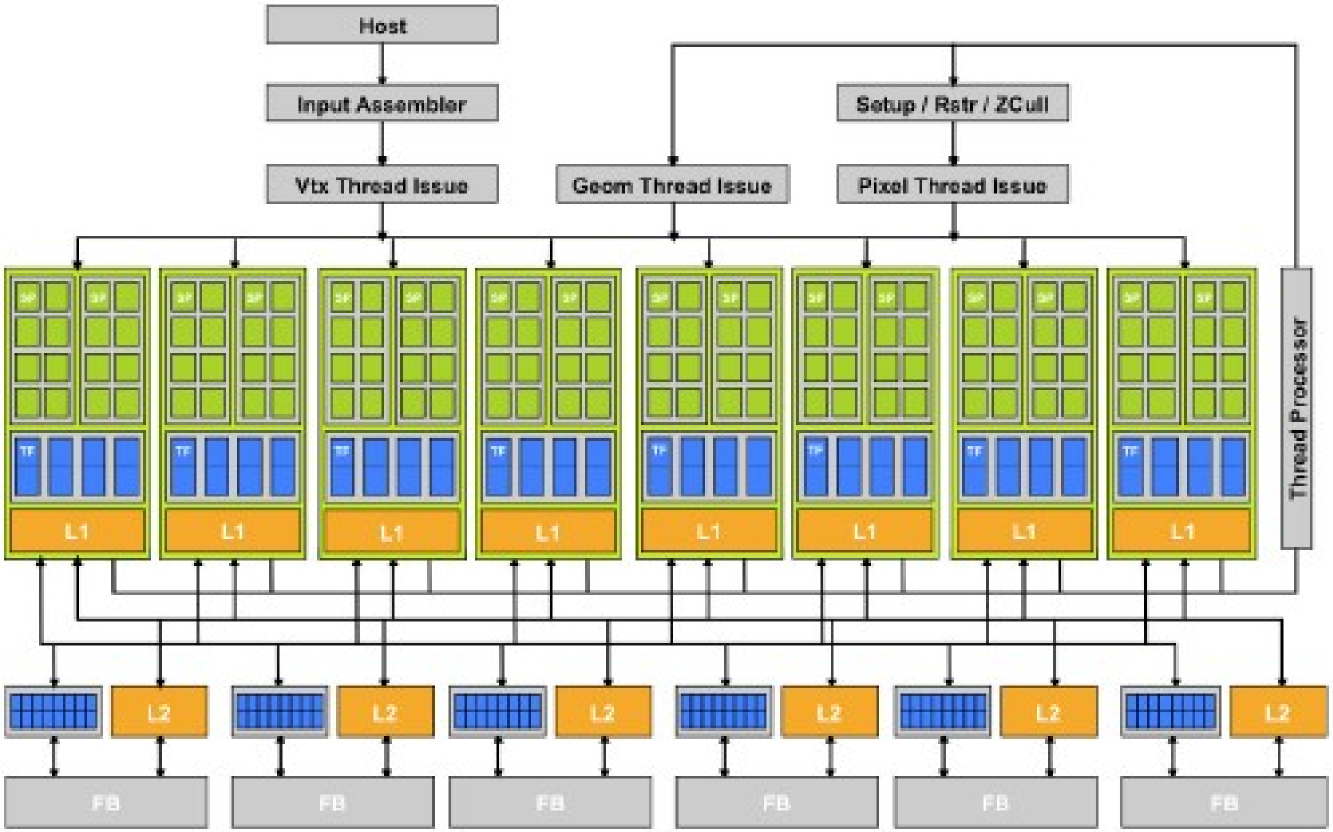
- Task parallelism
  - Enqueue multiple kernels to run in parallel
- Kernel and thread synchronization
- Reading and writing to images
  - Interoperability with OpenGL
- Performance issues
  - Coalescing memory accesses

References

- OpenCL Programming Guide for the CUDA Architecture (www.nvidia.com)
- NVIDIA OpenCL JumpStart Guide (www.nvidia.com)
- Tom R. Halfhill, Looking Beyond Graphics, White paper
- David Patterson, The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges, White paper
- Peter N. Glaskowsky, NVIDIA's Fermi: The First Complete GPU Computing Architecture, White paper
- Aaftab Munshi, OpenCL, Parallel Computing on the GPU and CPU, SIGGRAPH 2008: Beyond Programmable Shading(presentation)

Thanks!
Questions?

Backup



GeForce 8800