Introduction to General Purpose GPU Computing

Xiaoqing Tang

University of Rochester

March 16, 2011

Fun news

Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU

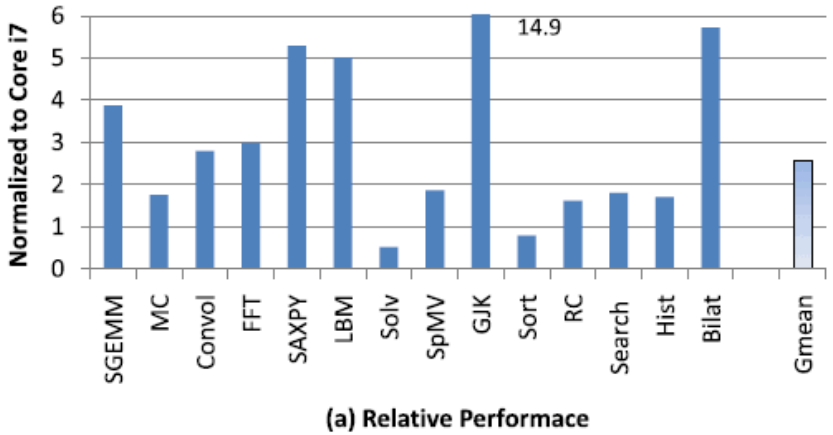


Figure 1: Comparison between Core i7 and GTX280 Performance.

Contents of Today's Talk

- Introduce GPUs and GPGPU computing
  - Execution model
  - Memory model
  - Programming model
  - Code examples
- Discuss what applications are appropriate for GPUs
- Advantages and disadvantages

What is a GPU?

- GPU=Graphics Processing Unit
- Processes 3D graphics/videos, render pixels, and send them to the monitor(s).

Birth of GPUs

- Motivation: Massive geometric transformations, e.g. vertex transformation, rasterization, global illumination.
- Problem: The amount of computation is so huge that overwhelms CPU.
- Algorithm feature: Same algorithm works on different sets of data. (Heavy data parallelism)
- Solution: Design a special dedicated processing unit to carry out this huge computation. This unit focuses on the data level parallelism.

Birth of GPGPU Computing

- Programmable shaders (vertex shaders, geometry shaders, pixel shaders)
- Program not only for graphics applications on shaders: GPGPU computing.
- Unified shaders even simplifies GPGPU computing.

Basic Architecture of GPUs: Execution Model

- Usually serves as a PCIe device on a PC.
- GPU programs (kernels) are sent to GPU through PCIe bus, the as way as it's done to data.
- Vendors provide APIs to communicate with GPUs.

Basic Architecture of GPUs: Execution Model

- Single-Instruction-Multiple-Data (SIMD) architecture.
  - One instruction is run over multiple pieces of data, typically a multiple of 32, i.e., 1 PC for 32 pieces of data.
  - A program can execute different instructions in parallel, but each instruction must have a multiple of (typically) 32 pieces of data to maximize performance.
  - Result: More ALUs available on the chip of the same size.
- Hardware multithreading
  - Typically the number of threads GPU maintains is bigger than the number of cores. ($>$10x)
  - Prevent stalling as much as possible if a group of threads is blocked by either memory latency or instruction dependency.
  - Hardware context switch to handle thousands of threads: a huge number of registers to store all threads, manipulate intermediate pipeline instruction/data.
  - Result: Hide memory access latency and instruction pipeline latency.

Basic Architecture of GPUs: Execution Model

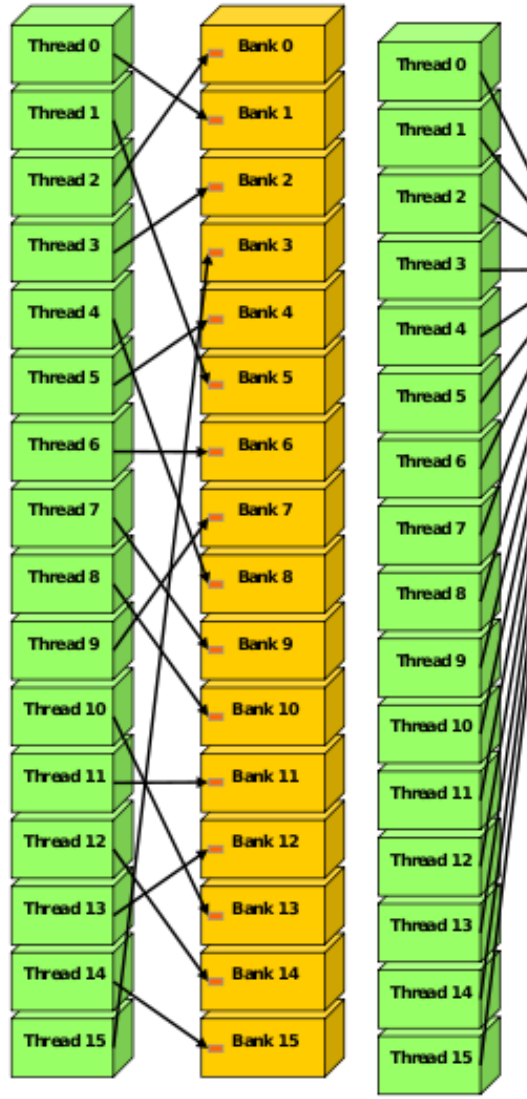
- Obviously speeds up the computation of the same instruction on multiple data
- But what about condition jumps? Different branches for the 32 pieces of data?
- Current solution: Will run both branches, but the data which does not go into the branch will temporarily be inactive.

Basic Architecture of GPUs: Memory Model

- GPU memory is optimized for bandwidth
  - GPUs (especially high-end GPUs) are equipped with GDDR memory.
  - More channels to gain a larger bus size (up to 8 channels each of 64 bits), increasing bandwidth.
  - Some technical changes which further increases bandwidth. (8n prefetch, request/receive data at the same cycle, etc.)
  - GDDR5 vs. DDR3: 12x theoretical bandwidth (170 GB/s under 6 channels, vs. 15 GB/s under dual channels).
- Memory latency under heavy parallelism
  - Synchronization of memory access across multiple threads causes latency
  - Latency is hidden by hardware context switch

Basic Architecture of GPUs: Memory Model

- What memory access pattern is good? What is bad?
- Prevent making memory access on global memory.
- Shared memory access?
  - Shared memory is divided into 16 or 32 banks.
  - Conflict-free if no bank conflict
  - Conflict-free if all threads access the same bank (cache).



Computing with GPUs: Programming Model

- A lot easier than before. Unified shader takes place of vertex/pixel/geometry shader so that programmers don't need to program under 3 different models.
- nVidia provides CUDA SDK to program using a subset of C/C++ along with some extensions.
- Emerging open standard OpenCL, supported by major GPU vendors like nVidia, AMD and Intel.

Programming for GPUs in C: Minimal Nontrivial Example

CUDA solution:

```
int a[1024],b[1024],c[1024];

__global__ void add(int* a, int* b, int* c) {
    int tid = blockIdx.x;
    c[tid] = a[tid] + b[tid];
}

int main() {
    int i;
    int *dev_a, *dev_b, *dev_c;
    for(i=0; i<n; ++i)
        a[i] = b[i] = i;
    cudaMalloc((void**)&dev_a, sizeof(a));
    cudaMalloc((void**)&dev_b, sizeof(b));
    cudaMalloc((void**)&dev_c, sizeof(c));
    cudaMemcpy(dev_a, a, sizeof(a), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, sizeof(b), cudaMemcpyHostToDevice);

    add<<<1024,1>>>(dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, sizeof(c), cudaMemcpyDeviceToHost);
}
```

Programming for GPUs in C: Minimal Nontrivial Example

OpenCL solution:

```
...//variable definitions omitted

__kernel void add(__global int* a, __global int* b, __global int* c) {
    int tid = get_global_id(0);
    c[tid] = a[tid] + b[tid];
}

int main()
{
    context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL); //create context
    ...//get device list, code omitted
    cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
    memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(a), a, NULL);
    ...//other 2 clCreateBuffer are omitted
    program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL); //create program
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL); //build program
    kernel = clCreateKernel(program, "add", NULL); //create kernel
    clSetKernelArg(kernel, 0, (void*)&memobjs[0], sizeof(cl_mem)); //set arguments for kernel
    ...//other 2 clSetKernelArg are omitted

    global_work_size[0] = 1024;
    clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL); //run

    clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0, sizeof(c), c, 0, NULL, NULL);
}
```

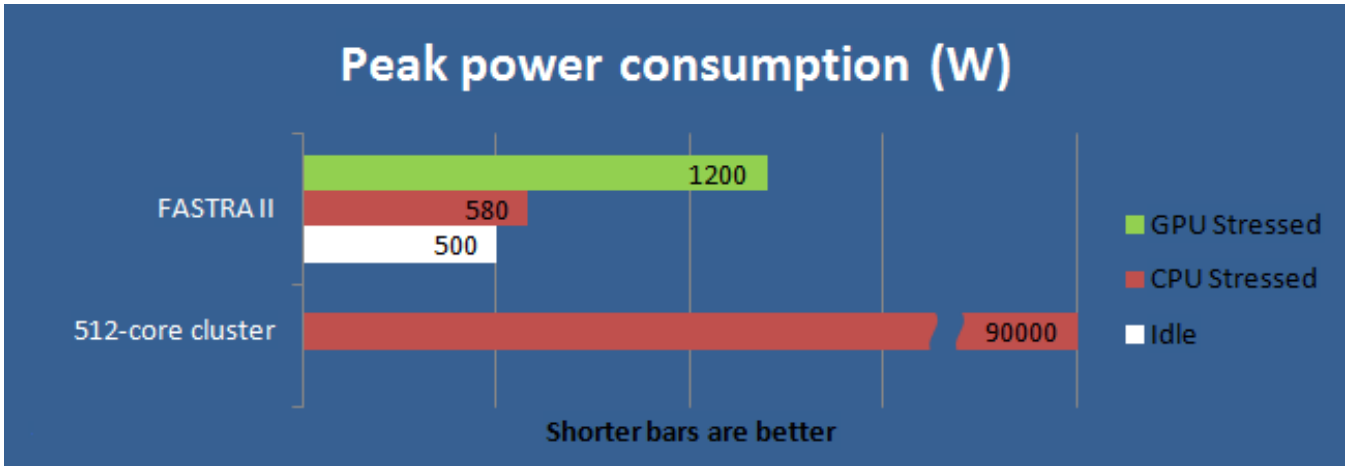
Programs Appropriate to Run on GPUs

- Characterization
  - Fewer branches
  - Memory access patterns to prevent bank conflicts
- Examples that can easily run on GPUs well
  - 3D graphics processing based on rasterization
  - FFT (enables a lot of applications to have performance gain)
- Examples that can *not* easily run on GPUs well
  - Compiler, in particular scanner and parser
  - Ray tracing (active research area)

Advantage and Disadvantage of GPU Computing

Advantage:

- Fast and Cheap
  - Newest nVidia GTX 580 delivers a theoretical 1.5 TFlops at $500. How much per TFlops? How much per TFlops for CPUs?
- Energy efficient



Power consumption of FASTRA2 (6x GTX 295 and 1x GTX 275, 12TFlops) and a 512-core normal cluster.

Advantage and Disadvantage of GPU Computing

Disadvantage:

- Not all algorithms can have theoretical speedup.
- Hard to program.
- No mature industrial/academic standard model.
  - Architecture is still developing fast.
  - Ray tracing can potentially develop a completely new graphics rendering pipeline.

Thanks!