Heterogeneous computing and future computer architectures

Hung-Wei Tseng
Limited CPU performance improvements
Outline

- The changing roles of GPUs
- Storage devices
- Future perspective
The changing role of GPUs
GPU (Graphics Processing Unit)

- Originally for displaying images
- HD video: 1920*1080 pixels * 60 frames per second
- Graphics processing pipeline

These shaders need to be “programmable” to apply different rendering effects/algorithms (Phong shading, Gouraud shading, and etc...)
Basic concept of shading

They are all “vectors”

\[
\begin{align*}
I_{amb} &= M_{amb} \cdot I_{amb} \\
I_{diff} &= K_{diff} \cdot M_{diff} \cdot (N \cdot L) \\
I_{spec} &= K_{spec} \cdot M_{spec} \cdot (R \cdot V)^n \\
I_{total} &= I_{amb} + I_{diff} + I_{spec}
\end{align*}
\]

For each “point/pixel”

```c
void main(void)
{
    // normalize vectors after interpolation
    vec3 L = normalize(o_toLight);
    vec3 V = normalize(o_toCamera);
    vec3 N = normalize(o_normal);

    // get Blinn-Phong reflectance components
    float Iamb = ambientLighting();
    float Idif = diffuseLighting(N, L);
    float Ispe = specularLighting(N, L, V);

    // diffuse color of the object from texture
    vec3 diffuseColor = texture(u_diffuseTexture, o_texcoords).rgb;

    // combination of all components and diffuse color of the object
    resultingColor.xyz = diffuseColor * (Iamb + Idif + Ispe);
    resultingColor.a = 1;
}
```
An Overview of Kepler GK110 and GK210 Architecture

Kepler GK110 was built first and foremost for Tesla, and its goal was to be the highest performing parallel computing microprocessor in the world. GK110 not only greatly exceeds the raw compute horsepower delivered by previous generation GPUs, but it does so efficiently, consuming significantly less power and generating much less heat output.

GK110 and GK210 are both designed to provide fast double precision computing performance to accelerate professional HPC compute workloads; this is a key difference from the NVIDIA Maxwell GPU architecture, which is designed primarily for fast graphics performance and single precision consumer compute tasks.

While the Maxwell architecture performs double precision calculations at a rate of 1/32 that of single precision calculations, the GK110 and GK210 Kepler-based GPUs are capable of performing double precision calculations at a rate of up to 1/3 of single precision compute performance.

Full Kepler GK110 and GK210 implementations include 15 SMX units and six 64-bit memory controllers. Different products will use different configurations. For example, some products may deploy 13 or 14 SMXs.

Key features of the architecture that will be discussed below in more depth include:

- The new SMX processor architecture
- An enhanced memory subsystem, offering additional caching capabilities, more bandwidth at each level of the hierarchy, and a fully redesigned and substantially faster DRAM I/O implementation.
- Hardware support throughout the design to enable new programming model capabilities

GK210 expands upon GK110’s on-chip resources, doubling the available register file and shared memory capacities per SMX.

SMX (Streaming Multiprocessor)
Inside each SMX Streaming Multiprocessor (SMX) Architecture

The Kepler GK110/GK210 SMX unit features several architectural innovations that make it the most powerful multiprocessor we've built for double precision compute workloads.

- SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).
Once the pixels fragments in a tile have been shaded, they flow to the Render Back-Ends (RBEs). The RBEs apply depth, stencil and alpha tests to determine whether pixel fragments are visible in the final frame. The visible pixels fragments are then sampled for coverage and color to construct the final output pixels.

The RBEs in GCN can access up to 8 color samples (i.e. 8x MSAA) from the 16KB color caches and 16 coverage samples (i.e. for up to 16x EQAA) from the 4KB depth caches per pixel. The color samples are blended using weights determined by the coverage samples to generate a final anti-aliased pixel color. The results are written out to the frame buffer, through the memory controllers.

The graphics pipeline is orchestrated using the same set of techniques as the ACEs. Each stage of the 3D pipeline can operate concurrently, as can any ACEs. The primitive and pixel pipelines are connected to the programmable GCN shaders through crossbar fabrics. The task queues synchronize different shaders and fixed function hardware through cache or memory.

The advantage of GCN’s flexibility is evident in the first few products that have scaled across all four dimensions. The AMD Radeon™ HD 7970 splits the screen into 2 primitive pipelines and 4 pixel pipelines, with 32 compute units for shading and a 384-bit memory interface. The GCN pixel pipelines are organized into 2 RBEs and 3 memory controllers, a 50% boost in memory bandwidth. In contrast, the AMD Radeon™ HD 7770 GHz Edition has a single primitive pipeline, 2 pixel pipelines and 10 compute units. The pixel pipelines in the AMD Radeon™ HD 7770 GHz Edition also scaled back to 2 memory controllers, for a 128-bit wide interface.

Figure 7: AMD Radeon™ HD 7970
Another crucial innovation in GCN is coherent caching. Historically, GPUs have relied on specialized caches (such as read-only texture caches) that do not maintain a coherent view of memory. To communicate between cores within a GPU, the programmer or compiler must insert explicit synchronization instructions to flush shared data back to memory. While this approach simplifies design, it increases overhead for applications which share data. GCN is tailored for general purpose workloads, where algorithms that communicate between cores are common. The cache coherency protocol shares data through the L2 cache, which is significantly faster and more power efficient than using off-chip graphics memory.

In tandem with cache coherency, GCN introduces virtual memory through a combination of hardware and driver support. Virtual memory eliminates the most challenging aspects of memory management and opens up new capabilities. AMD's unique expertise in both high performance graphics and microprocessors was particularly beneficial, as GCN's virtual memory model has been carefully defined to be compatible with x86. This simplifies moving data between the CPU and the discrete GPU in initial products. More importantly, it paves the way for a single address space that is seamlessly shared by CPUs and GPUs.

Sharing, rather than copying, data is vital for performance and power efficiency and a critical element in heterogeneous systems such as AMD's Accelerated Processing Units (APUs).
Programming GPGPU

- A GPGPU application contains the “host program” and “GPU kernels”
- Host program: A C/C++ based CPU program that invokes GPU API
- GPU kernels: C/C++-like programs running on GPUs
- Programming models
  - CUDA (Compute Unified Device Architecture)
    - Proposed by NVIDIA
    - Only NVIDIA GPUs support
  - OpenCL
    - Maintained by Khronos Group (non-profit)
    - Supported by Altera, AMD, Apple, ARM Holdings, Creative Technology, IBM, Imagination Technologies, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS
What a host program looks like?

```c
int main (int argc, const char * argv[]) {
    dispatch_queue_t queue = gcl_create_dispatch_queue(CL_DEVICE_TYPE_GPU, NULL);

    float *a = (float *)malloc(ARRAY_SIZE*ARRAY_SIZE*sizeof(cl_float));
    float *b = (float *)malloc(ARRAY_SIZE*ARRAY_SIZE*sizeof(cl_float));
    float *c = (float *)malloc(ARRAY_SIZE*ARRAY_SIZE*sizeof(cl_float));

    if(a == NULL || b == NULL || c == NULL)
        fprintf(stderr, "allocating array c failed\n");
    else init(a, ARRAY_SIZE); init(b, ARRAY_SIZE);

    void *gpu_a = gcl_malloc(sizeof(cl_float) * ARRAY_SIZE*ARRAY_SIZE, a,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR);
    void *gpu_b = gcl_malloc(sizeof(cl_float) * ARRAY_SIZE*ARRAY_SIZE, b,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR);
    void *gpu_c = gcl_malloc(sizeof(cl_float) * ARRAY_SIZE*ARRAY_SIZE, NULL,
        CL_MEM_WRITE_ONLY);

    dispatch_sync(queue, ^{
        cl_ndrange range = {2, {0, 0, 0},{ARRAY_SIZE, ARRAY_SIZE, 0},{16, 16, 0}};
        matrix_mul_kernel(&range,(cl_float*)gpu_a,(cl_float*)gpu_b,(cl_float*)gpu_c,
            (cl_int)ARRAY_SIZE);
        gcl_memcpy(c, gpu_c, sizeof(cl_float) * ARRAY_SIZE*ARRAY_SIZE);
    });
    gcl_free(gpu_a); gcl_free(gpu_b); gcl_free(gpu_c);
    dispatch_release(queue);
    free(a); free(b); free(c);
    return 0;
}
```

**Initialize GPU runtime**

**Initialize host memory objects**

**Allocate GPU memory space & copy data to GPU memory**

**Submitting GPU kernel to the runtime**

**Copy data from GPU memory to main memory**
What a GPU kernel looks like?

```c
__kernel void matrix_mul(__global float* input_a,
                         __global float* input_b,
                         __global float* output,
                         int size)
{
    int col = get_global_id(0);
    int row = get_global_id(1);

    float value = 0;
    for (int k = 0; k < size; ++k)
    {
        value += input_a[row * size + k] * input_b[k * size + col];
    }
    output[row * size + col] = value;
}
```

Identify the current GPU thread.
Demo

- Matrix multiplication
- block algorithm v.s. naive OpenCL code
How things are connected
New overhead/bottleneck emerges

GPU

CPU

PCIe Switch

DRAM

Second Storage Devices
APU (Accelerated Processing Unit)

- Having GPU cores and CPU cores within the same chip
- It’s now very common in intel and AMD lineups
What about storage devices?
Hard Disk

- Position the head to proper track (seek time)
- Rotate to desired sector. (rotational delay)
- Read or write data from/to disk to in the unit of sectors (e.g. 512B)
- Takes at least 5ms for each access
Solid State Drives

- Using flash memory chips to store data
- Must be read/written in the unit of a “flash page” (4KB or 8KB depending on the chip)
  - Reading a page takes \textbf{100us} \checkmark \textbf{Good!}
  - Writing/programming a page takes less than \textbf{2ms} \textbf{~~ OK}
- Must be erased in the unit of a “flash block”
  - A flash block contains 128 - 384 pages (0.5MB - 3MB)
  - A flash page cannot be reprogrammed/rewritten if the block is not erased yet
  - Erasing a block takes several (less than 5, usually) \textbf{ms} \textbf{Oh! No!}
  - A flash block can only be erased for limited amount of times
    - SLC: less than 100,000 times, faster
    - MLC: less than 10,000 times
FTL (Flash translation layer)

- We are always lazy to modify our applications
- FTL maintains an abstraction of LBAs (logic block addresses) used between hard disk drives and software applications
- FTL dynamically maps your logical block addresses to physical addresses on the flash memory chip
- FTL performs copy-on-write when there is an update
- FTL reclaims invalid data regions and data blocks to allow future updates
- FTL executes wear-leveling to maximize the life time
- It needs your SSD to have a processor in it now
How FTL works

SSD Processor

<table>
<thead>
<tr>
<th>LBA</th>
<th>Flash Block</th>
<th>Flash Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3241</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x3242</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>0x3243</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0x3244</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0x3245</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0x3246</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

valid page  invalid page  free page

block #0  block #1  block #2  block #3  block #4
How FTL works

You have to erase the block, write 0,15 to somewhere else before you can write this again!
Writing: 2ms, erasing 3ms, no faster than H.D.D.
Garbage collection

SSD

<table>
<thead>
<tr>
<th>LBA</th>
<th>Flash Block</th>
<th>Flash Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3241</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>0x3242</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>0x3243</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0x3244</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0x3245</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0x3246</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

block #0

block #1

block #2

block #3

block #4

- valid page
- invalid page
- free page
Future perspective
Challenges still

- Moore’s law is slowing down
- Leakage power is constraining the processing power of a single chip
New opportunities

Processors are everywhere in your computers
Can we writing/executing programs from where we store data? Programs’ viewpoint today: from CPU.
Data-centric computing

Programming models, OS, runtime systems, compilers

Efficient processors
Workloads
Conclusion

• In the past, software engineers and hardware engineers work on different side of the ISA abstraction
  • Software engineers have no idea about what happen in processors
  • Hardware engineers have no sense about what application wants
  • This works fine if we can keep accelerating CPUs, but not true anymore

• We need new execution & programming model to better utilize these emerging hardware components
Announcement

• Final review this Thursday
• Will have a small reward & performance session (less than 30 minutes) after our final
  • Pizza provided