GPUs and GPU Programming

1 Contemporary GPU System Architecture

1.1 Historical Context

Up until 1999, the GPU did not exist. Graphics on a personal computer was performed by a video graphics array (VGA) controller, sometimes called a graphics accelerator. A VGA controller was a combination of a memory controller and a display generator with attached DRAM. As it became less expensive to build more powerful processors, it also became less expensive to add more functionality to these graphics accelerators. In the 1990's, VGA controllers began to incorporate some three-dimensional (3D) functions, such as triangulation, rasterization, and texture mapping and shading. These operations are described below.

The availability of these more powerful graphics chips led software makers to create software that used these chips, fueled by the public's insatiable desire for life-like real-time 3D graphics, and the demands of the computer gaming industry, the movie industry, and the television industry. Chip manufacturers completed this cycle by responding in turn with more powerful chips. Eventually, this cycle of growth resulted in graphics controllers that had as much processing power as the CPU itself, although their limited purpose made them unsuitable to be used as CPUs.

The first graphics processing unit (GPU), NVIDIA's GEForce 256, appeared in 1999. In addition to the operations that had become standard by then, this chip incorporated functions to perform in hardware transforms (movement in 3D), lighting and shading (altering the color of surfaces of the scene based on lighting information.) In November 2006, NVIDIA's GeForce 8800 gave birth to their new GPU Computing model. The GeForce 8800 was based on the G80 architecture and brought several key innovations to GPU computing. The G80 series was the largest commercial GPU at the time, containing approximately 686 million transistors. By 2008, eight generations of the GEForce had been built, some of which provided full support for 3D graphics libraries such as Direct3D.

Over the next few years, the GPUs became more and more programmable, replacing fixed function dedicated logic by programmable processors. Integer arithmetic was replaced by floating-point arithmetic, and the degree of parallelism within the chips increased dramatically. It was not long before chip manufacturers started adding instructions and memory to the GPUs so that they could support general purpose programming, and this turned them into fully general-purpose processors, known as GPGPUs. At this point the GPGPU is a processor with unprecedented floating-point performance and programmability.

1.2 Differences Between GPUs and CPUs

GPUs do not fit into any classification scheme. Parts of it are MIMD and parts of it are SIMD. The NVIDIA GeForce 8800 GTX, for example has 16 identical Tesla processors, each of which is a sort of multiprocessor. Thus, from a large scale view, it is MIMD. Each Tesla processor consists of eight streaming processors, with a SIMD architecture. Each streaming processor executes the same instruction on different data. Within the processor there is a shared memory that each streaming processor can access.

Although the hardware is SIMD, the Tesla programmer interface creates the illusion that the multiprocessor is MIMD. It achieves this by making certain threads inactive when they are not supposed to execute a given instruction, and by high level parallel fine-grained multithreading. At its best, all threads are busy all of the time. The hardware is more like SIMT – single instruction multiple thread, than SIMD. But if the programmer does not write the code carefully, the machine will not take advantage of the maximum parallelism possible.

A GPU is a multiprocessor, sometimes containing hundreds of processors. The intended purpose of a GPU is to perform graphics operations, which is what they do well, but they can be used for other computations as well. So how are they different from CPUs at this point?

- GPUs do not perform all of the operations that a CPU can perform; their instruction sets are narrowly focused on graphics acceleration.
- The programming interfaces to GPUs are high-level application programming interfaces such as *OpenGL*, *Cg*, and *DirectX*, together with high-level graphics shading languages such as *C for Graphics* (*Cg*) and the *High Level Shader Language* (*HLSL*). These languages are supported by compilers that generate intermediate languages, which are optimized by the specific GPU driver software, which generates the the specific machine instructions for the GPU.
- Graphics processing includes many stages in a pipeline of operations, including vertex shading, geometry shading, rasterization, and pixel shading. These operations, described below, are performed on a massively parallel scale in a pipe-lined fashion.
- Vertices can be drawn independently, and pixel fragments can be rendered independently. This independence allows the computation to proceed using many independent and parallel threads of control.
- GPUs are designed to work well on 4-tuples. This is because a vertex in three dimensions is represented by a set of four coordinates, (x, y, z, w), called ***homogeneous coordinates***. The fourth coordinate, w, is used to facilitate projecting the 3D point into two-dimensions in a way that creates the illusion of depth (known to the artist as perspective drawing and to the mathematician as projective geometry.) Also, having the fourth coordinate makes the basic transformations of rotation, translation, and scaling obtainable by matrix multiplication. Pixels consist of four coordinates also, from a color space with an alpha channel, (r, g, b, alpha). Vertices and pixels each consist of four 32-bit floating point numbers.
- Unlike general purpose applications, graphics computations have working sets that can be hundreds of megabytes.
- There is much more data parallelism in graphics applications than general purpose applications.

A consequence of these differences is that

- GPUs do not rely on multi-level caches to overcome long latencies to memory. Instead, they rely on having enough threads to hide the latency and they use multithreading.
- GPUs rely on extensive parallelism to obtain high performance.
- GPU main memory is designed for high bandwidth rather than small latency. GPU's have smaller memories than CPUs.
- GPU processors are multithreaded and each is highly parallel. In the past, they had specialized processors for each stage of the pipeline, but more of them now have homogeneous processors.
- GPUs used to have four-stream SIMD processors within them, but now these are being replaced by regular processors and scalar instructions.
- GPUs have no need for double-precision floating point instructions, but as they become used more and more for non-graphical processing, this functionality is being added to them.

There is a growing base of general purpose applications that have been ported to GPUs. The term general purpose GPU, or GPGPU refers to a method of using GPUs for non-graphics applications. NVIDIA has developed a programming language, CUDA (Compute Unified Device Architecture) that enables programmers to write C code for GPUs.

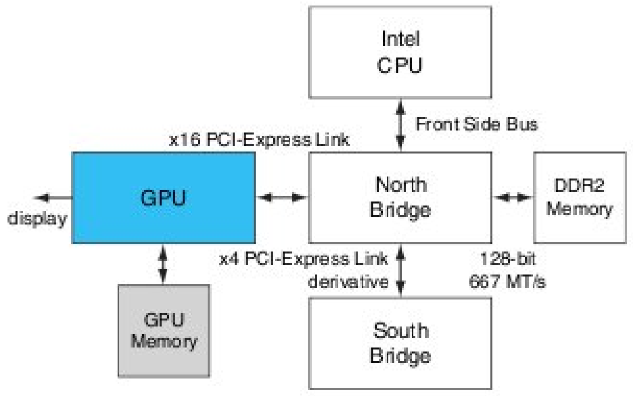

Fig. 1: Intel interconnects in the CPU

1.3 The GPU in Relation to the CPU

To get a perspective on the position of the GPU within a typical computer, see Figure 1. In Intel processors, the GPU is connected via the PCI-Express link to the North Bridge in the Intel chip-set, and the GPU's *device memory*, not accessible to the rest of the chip, is connected directly to it. Applications that run in the CPU do not have access to this GPU device memory, but a program specifically written to run on a GPU can take advantage of it.

Before we look at the GPU architecture in detail, it is best to explain briefly the various graphics operations that are performed in graphics processing.

1.4 The Logical 3D Graphics Pipeline

1.4.1 The Input

It will help to understand why the GPU architecture is what it is if you have a basic understanding of what must be done to render a 3D image to the screen, with all of the lighting effects and texturizing that you see in modern day applications. The most demanding application of a graphics processor is to perform real-time, high-resolution 3D image processing at a frame rate no less than 60 frames per second, as is necessary to create computer-based animations. To make this possible, the graphics processor must be able to exploit the high degree of data-parallelism in the problem.

A 3D image is represented by a 3D model, which is a collection of 3D points. Usually, 3D models start out as triangulated surfaces, as shown in Figure 2. The vertices of the triangles define the shape and are the starting point for display of the object. Thus, a 3D shape begins as a set of these vertices.

Graphics processing proceeds as a sequence of pipelined stages. The basic graphics pipeline is shown in Figure 3. The individual stages are described below.

1.4.2 Z-Buffers

One of the major tasks in three-dimensional (3D) graphics processing is determining where each 3D point should be placed on the 2D screen (which is essentially a projection problem), and what color it should have. A key concept underlying the projection problem is how to determine which 3D points are visible and which are obscured by other points, which is the ***visibility problem***.

In computer graphics, ***z-buffering*** is the management of image depth coordinates in three-dimensional (3D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem,

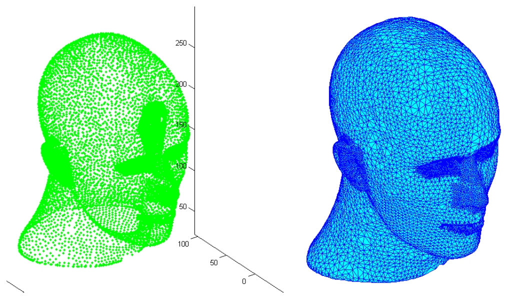

Fig. 2: Triangulated representation of a head

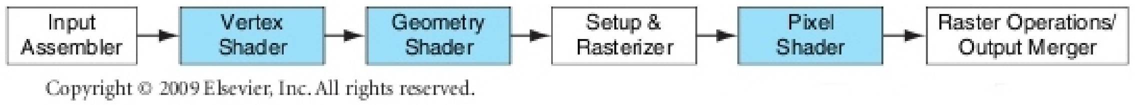
Input Assembler
Vertex Shader
Geometry Shader
Setup & Rasterizer
Pixel Shader
Raster Operations/ Output Merger
Copyright © 2009 Elsevier, Inc. All rights reserved.

Fig. 3: Graphics Logical Pipeline

which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. The ***painter's algorithm*** is another common solution which, though less efficient, can also handle non-opaque scene elements. Z-buffering is also known as ***depth buffering***.

An arbitrary point on a surface in 3D is represented by the coordinates (x,y,z), where z is the depth coordinate. Initially all depth coordinates are normalized, varying between 0 and 1, with 1 being the furthest distance from the viewing plane. When an object is rendered by a 3D graphics card, the depth of a generated pixel (its z coordinate) is stored into the z-buffer (or depth buffer). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. The intensity value for that pixel (its color and alpha channel) are stored in a parallel 2D buffer called the ***refresh buffer***.

When multiple objects in the scene must be rendered in the same pixel, the graphics card must decide which object's pixel is closest to the observer. The chosen depth is then saved to the z-buffer, replacing the old one, and its intensity replaces the current value of the intensity at that pixel in the refresh buffer. In the end, the z-buffer will allow the graphics card to correctly reproduce the usual depth perception: a close object hides a farther one. This is called ***z-culling***.

Determining which pixel is closest is a calculation based upon the planar surfaces of the 3D object. When a surface is triangulated, every point is either a vertex of a triangle or a point inside the triangle formed by the three vertices. Each triangle defines a plane, whose equation can be obtained by solving a system of linear equations using those three sets of coordinates. This leads to a planar equation of the form $Ax + By + Cz = D$. where A, B, C, and D are constants. Each point in that plane satisfies this equation. By algebraic manipulation, if x and y are known, z can be obtained from the equation

$$z = \frac{-Ax - By + D}{C}$$

The z-values so obtained lie between 0 and 1.0.

For efficiency, the depth-buffering algorithm will process each scan line (horizontal line of pixels) in succession. Two adjacent pixels differ only in their x-coordinate: (x, y) is followed by$(x + 1, y)$. If the depth value for (x, y) is z, then the depth value z' for $(x + 1, y)$ can be obtained by a single subtraction operation, since

$$z' = \frac{-A(x+1) - By + D}{C} = \frac{-Ax - By + D}{C} - \frac{A}{C} = z - \frac{A}{C}$$

and A/C is a stored constant for each plane.

1.4.3 The Pipeline Stages

Input Assembler The ***input assembler*** receives the 3D representation of the scene as a collection of geometric primitives such as points, lines, and vertices. It then distributes the vertices to the vertex shader. The input assembler typically assembles vertices into several different primitive types such as line lists, triangle strips, or primitives with adjacency.

The input data may be a list of vertices, lines and so on, but it might also include an index buffer. Without an index buffer, a vertex might appear multiple times in the input to the graphics processor, because the vertex data was arranged in the order in which the larger primitives (e.g., lines, triangles) would be processed. Rather than storing the vertices themselves in an array, the array could contain references to vertices, which would be in a secondary structure. This array is an ***index buffer***. By using an index buffer, each vertex is stored exactly once, and references to it as a part of larger primitives are the indices into this array.

The input may also include a collection of ***textures*** that are to be applied to the scene to be displayed. You can think of a texture as a two-dimensional shape, usually a rectangle, containing colors or an image to be applied to a surface.

Vertex Shading Shader programs in general determine how lighting and shadows interact with the surfaces to be rendered. A ***vertex shader*** is a graphics processing function that maps vertices onto the screen and adds special effects to objects in a 3D environment by performing mathematical operations on the objects' vertex data. One of its purposes is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen, as well as a depth value for the Z-buffer, and then to apply color to it.

Examples of vertex shading effects include matrix palette skinning, which allows programmers to create realistic character animation with up to 32 "bones" per joint; deformation of surfaces, which gives developers the power to create realistic surfaces such as waves; and vertex morphing, which is used to morph triangle meshes from one shape to another, providing smooth skeletal animation. Vertex shaders are run once for each input vertex. Although vertex shaders can manipulate properties such as position, color, and texture coordinate, they cannot create new vertices. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present or the rasterizer otherwise.

Geometry Shading ***Geometry shading*** is the stage of the graphics pipeline after vertex shading. Its purpose is to enhance the details and accuracy of the 2D image by working at a larger degree of granularity than individual vertices. Its inputs consist of geometric primitives consisting of more than one vertex, such as lines and triangles. A geometry shader can take as its input, for example, the three points of a triangle and output intermediate points that can be used to refine the surface. It can only do this by operating with greater granularity. The geometry shader can modify the positions and orientation of the primitives.

Rasterization The word "raster" was originally used in the raster scan of cathode ray tubes (CRT), which paint the image line by line; the term is now used to mean a grid of pixels. ***Rasterization*** is a process for converting vectorized input to a bitmap form, i.e. a 2D array of pixels. Given a triangle, for example, represented by three vertices, it determines the locations of all pixels and pixel fragments that lie inside, or on the edge of, this triangle. There can be pixel fragments because a triangle is a mathematical object consisting of lines of infinitely small width that can intersect pixels rather than lie between them. Therefore, a single pixel can lie on each side of a line segment. The fragment of the pixel on the inside edge of a triangle's perimeter is a pixel fragment belonging to that triangle.

The rasterization stage actually does other processing as well. Typically it also does ***clipping***, meaning throwing away regions that are outside of the ***view frustum***. A frustum is a 3D shape that can be described as a 4-sided pyramid with its top lopped off. It is the 3D analog of a trapezoid. When you view a scene in perspective, you are looking at a frustum lying on its side usually. The front plane is the bottom of the pyramid, and the back plane is the lopped-off top. This is the view frustum.

The rasterization step typically does z-culling of pixels. As it is generating pixels, it can discover that some of them are behind others and should be culled. When this step is finished, what is left are the visible pixels and pixel fragments, represented by their screen positions and their depth values. These are passed to the pixel shader.

Pixel Shading A ***pixel shader*** is a function that computes the color and other attributes of each pixel or pixel fragment. Pixel shaders range from always outputting the same color, to applying a lighting value, to doing bump mapping, shadows, specular highlights, translucency and other phenomena. They can alter the depth of the pixel (for Z-buffering), or output more than one color if multiple render targets are active. A pixel shader alone cannot produce very complex effects, because it operates only on a single pixel, without knowledge of a scene's geometry or of neighboring pixels.

The following program, taken from the Patterson-Hennessey text, can give you an idea of the type of work that a pixel shader can do. It is written in *Cg*. Each pixel is run through this program.

```
void reflection(
    float2             texCoord       : TEXCOORD0,
    float3             reflection_dir : TEXCOORD1,
    uniform float      shiny,
    uniform sampler2D   surfaceMap,
    uniform samplerCUBE envMap,
    out float4         color          : COLOR )
{
    // Fetch the surface color from a texture
    float4 surfaceColor   = tex2D(surfaceMap, texCoord);

    // Fetch reflected color by sampling a cube map
    float4 reflectedColor = texCUBE(envMap, reflection_dir);

    // Output is a weighted average of the surface and reflected colors
    color = lerp(surfaceColor, reflectedColor, shiny);
}
```

The first input parameter, `texCoord`, is the (x, y) position of the pixel. This is used as an index into a texture map in order to apply a particular texture to that pixel. The texture map, `surfaceMap`, is a large 2D array that is passed to all pixels in the fourth parameter. The second input parameter, `reflection_dir`, is a 3D vector that represents the direction of the view with respect to the surface. To understand this, imagine that you could draw a line between the pixel and your eye. Next, rotate the plane in which the pixel lies so that it is the horizontal plane and then translate the plane so that the pixel is at the origin. Assume that the z-axis is upward and x and y are in the horizontal plane. The line from the pixel to your eye is a vector called the ***reflection direction***. It is the direction in which the light will bounce from the surface to your eye. The `envMap` parameter stores the colors that each face of a cube would have given the lighting conditions for the surface. Those colors are used to determine what color the reflected light would be, assuming the pixel did not absorb any of the light's spectrum.

The reflection function calls two functions, `tex2D()` and `texCUBE()` to compute the surface color and reflected color and then uses a weighted average of these two colors to return the value by which to color this particular pixel. The weighted average function is `lerp()`. Its third parameter is a floating point value that represents the weight to apply to each of the two colors. `lerp` is short for ***linear interpolation***, which, given two values a_1 and a_2 and a constant $0 \leq c \leq 1$, produces a value $(1 - c)a_1 + ca_2$. When $c = 0$, this evaluates to a_1 and when $c = 1$, it evaluates to a_2. For all values in between, the distance of the interpolated point from a_1 is proportional to c and hence the name. Linear interpolation is used more generally as follows. If

$f(x)$ is a function of x all of whose values are either hard to compute or unknown, and an approximation or estimate of it is needed, and two values are known, at, say at points x_1 and x_2, then a linear interpolation of $f(x)$ between the pair of points is given by $(1-c)f(x_1)+cf(x_2)$, $0 \le c \le 1$, which in effect treats the function as a linear function (a straight line) between the points. The approximation to f, denoted $\hat{f}(x)$, would be defined by letting $c = (x-x_1)/(x_2-x_1)$.

The Relation Between the Graphics Pipeline and the GPU

Different GPUs map the functions of the pipeline onto the hardware in different ways. Older GPUs had special purpose hardware units to implement each separate pipeline stage. Some of these stages are not programmable – they are fixed functions that do the same thing for all inputs. They are indicated in white in Figure 3. The other stages are programmable. Later GPUs have a single processor type that can be used to implement most of the pipeline stages. This is the case for the NVidia GEForce 8800 for example. These later GPUs use multiple, identical processors, each containing several independent functional units or even SIMD processors. As you will see though, some of the stages, such as input assembly and rasterization, remain in special purpose processors.

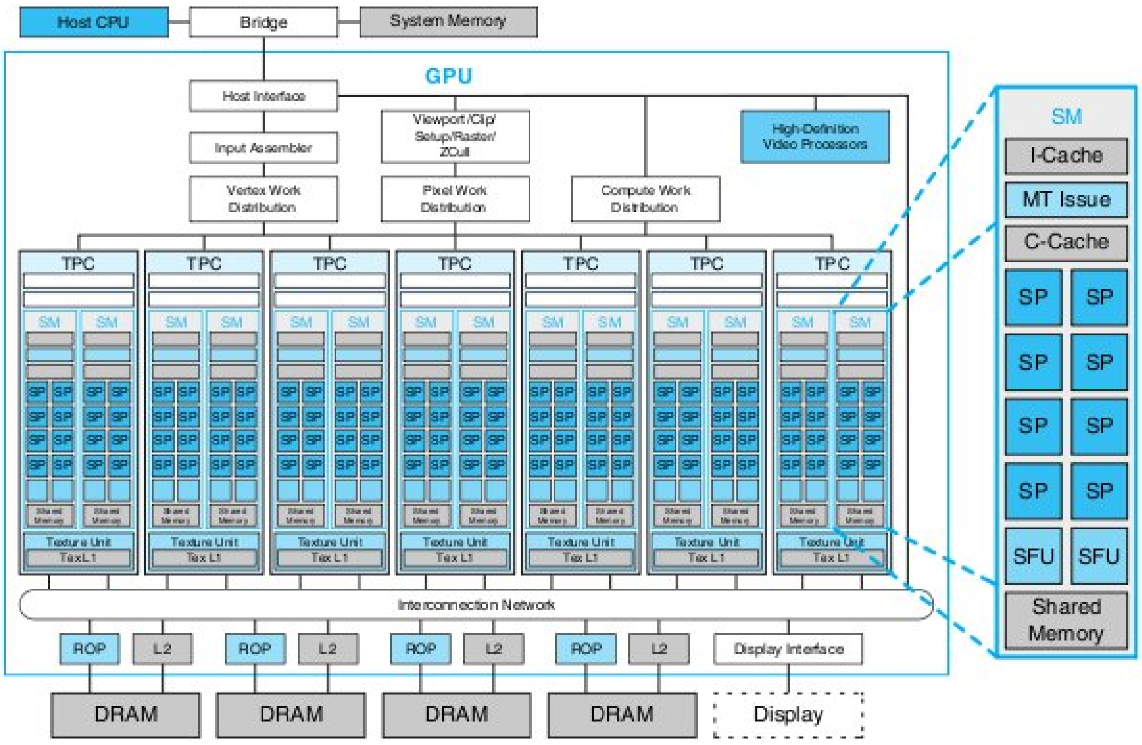

Fig. 4: Unified GPU architecture (GEForce 8800 w/ 112 SPs

1.5 Overview of the NVIDIA GPU Architecture

GPU terminology differs from conventional CPU language, and to make matters worse, NVIDIA changed its terminology over time, so we begin by defining the basic terms. We will use the GEForce 8800 GTX as an example architecture.

A ***discrete GPU*** is one that sits on a card that is plugged into the PCI-Express interconnect bus. In contrast, a ***motherboard-GPU*** is integrated into the chipset on the motherboard. Tesla-based GPUs can have from 1 to 16 nodes, each called a ***streaming multiprocessor*** (***SM***). NVIDIA uses the terms "node" and “streaming multiprocessor” interchangeably. The largest version available in 2008 was the *GeForce 8800 GTX*, with 16 SMs, each containing 8 multithreaded single-precision floating point and integer processing units, which used to be called ***streaming processors*** (***SP***) but are now called ***cores***. The clock rate was 1.35 GHz. The later Fermi-based GPUs support 32 cores per streaming multiprocessor, implying that they have up to 512 cores. The newest architecture, at the time of this writing, is the Kepler, which has up to 15 Streaming Multiprocessor (SMX) units, each of which has up to 192 single-precision CUDA cores, with each core having fully pipelined floating-point and integer arithmetic logic units. Figure 6 shows the architecture of a single Kepler SMX processor.

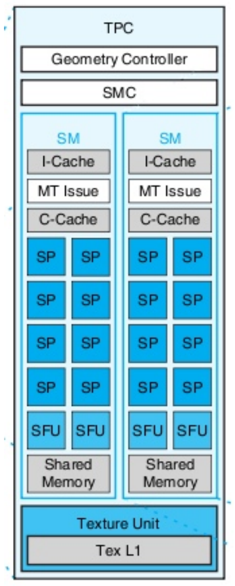

Fig. 5: The texture/processing cluster.

To give you some idea of the power of these GPUs, consider the fact that the *GEForce 8800 GTX* has a single-precision multiply-add instruction. This means that the add and multiply operations take place in a single instruction cycle. Given the preceding parameters, we can compute the peak performance of the 8800 GTX, i.e., the performance when all processors are kept busy all of the time:

```
Peak performance in GFLOPS/second = 16 SMs x (8 cores/SM)
                                     x 2 FLOPS/instruction (add and multiply) /core
                                     x 1 instruction/clock cycle
                                     x 1.35 x 10^9 clock cycles/second
                                   = 16 * 8 * 2 * 1.35*10^9 FLOPs/second
                                   = 345.6 GFLOPs / second
```

With the same clock speed, the Kepler GK110 would have a performance of almost 8 Teraflops!

Each of the 16 SMs of the GEForce 8800 has a local store with a capacity of 16KB as well as 8192 32-bit registers. The shared memory is partitioned into six partitions of 900 MHz Graphics DDR3 DRAM, each with an 8-byte wide datapath, with 128 MB per partition. Therefore, there is a total of 768 MB of memory, and the peak bandwidth is

```
Peak bandwidth (GB/second) = 6 x 8 bytes/transfer
                              x 2 transfers/clock cycle (double-data rate)
                              x 0.9 x 10^9 clock cycles/second
```

```
= 6 * 8 * 2 * 0.9 GB/second
= 86.4 GB/second
```

Figure4 depicts the architecture of an NVidia GEForce 8800 that has 14 streaming multiprocessors, distributed into 7 pairs of SMs. Each SM pair is integrated into a unit called a **texture/processor cluster** (TPC) that contains a shared geometry controller, a streaming multiprocessor controller (SMC), and a shared texture unit and cache and shared lines for load/store and I/O, as shown in Figure 5. The SMC controls access to the shared texture unit, load/store path, and I/O path. Each TPC is connected to an interconnection network that connects it to the device memory, L2 cache (containing textures), raster processors, and the interface to the actual display device, as well as to the bridge to the CPU and system memory.

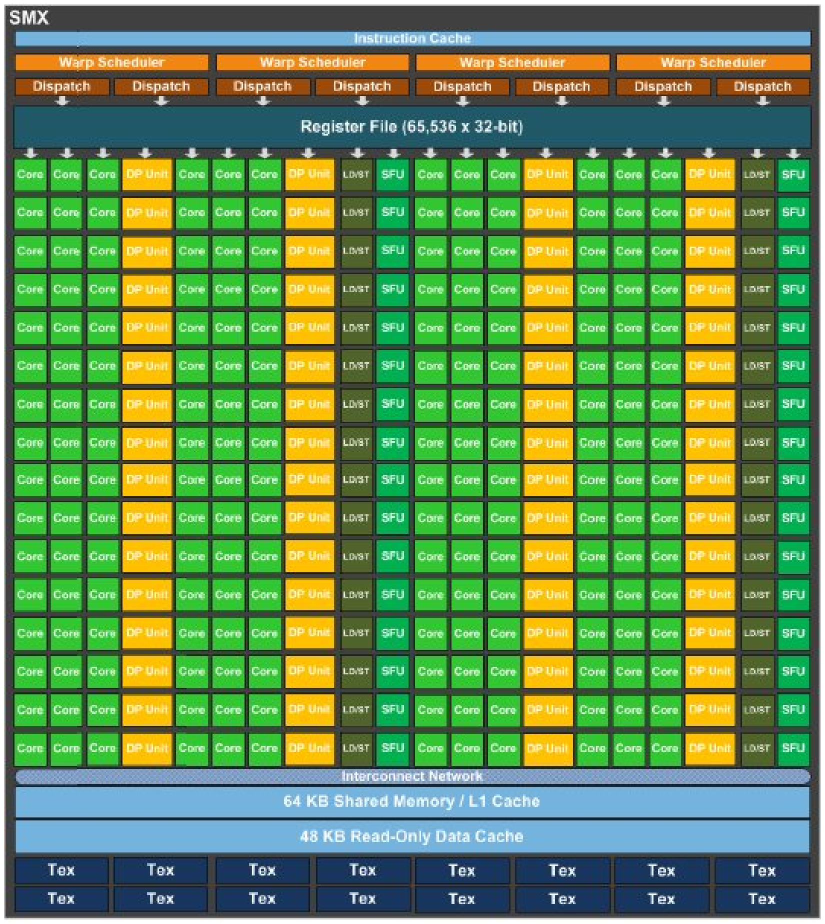

Fig. 6: The SMX architecture of the Kepler GK110 (*from the NVIDIA Kepler GK110 whitepaper.*)

Each SM in the Tesla GPUs contains 8 cores (32 in the Fermi GPUs), as well as its own local shared memory, an instruction cache, a constant cache, a multithreaded instruction unit, and two special function units (SFUs). The special function units compute special functions such as the transcendental functions (e.g., trigonometric functions), reciprocals, and square roots.

A core (SP) is the primary thread processor. In the GEForce 8800, each core is a multithreaded processor supporting 96 threads, with a register file containing 1024 scalar 32-bit registers for the use of these threads. The processor is fully pipelined and implements all 32-bit and 64-bit integer arithmetic, comparison, conversion, and logical PTX[1] instructions, as well as IEEE 754 standard single precision floating point operations

[1] PTX instructions are parallel thread execution instructions. Each such instruction is issued to multiple threads simultaneously.

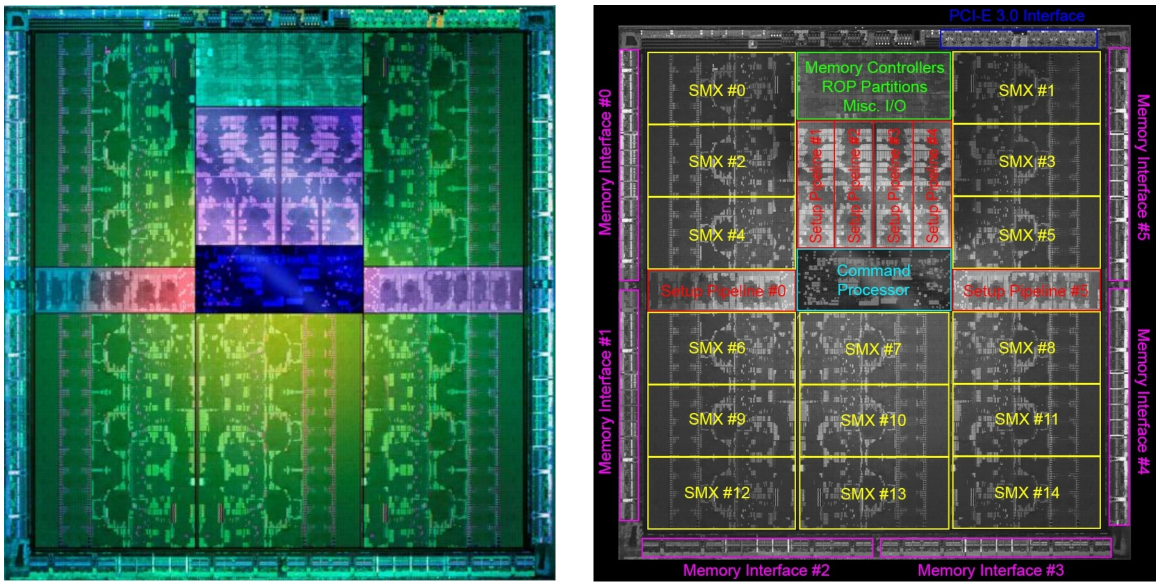

Fig. 7: Die photo of the Kepler GK110 with 15 SMX processors, and overlay showing locations.

and a compatible add/multiply operation. Earlier GPUs executed vector instructions, but the later GPUs were designed so that core processors executed ordinary scalar instructions.

1.6 Overview of the Computational Model

The GPU architecture is tightly coupled with the computational model; it is difficult to appreciate the architecture without knowing something about how programs are designed and what type of parallelism is provided in the computational model. Therefore, we turn our attention to how the GPU is programmed.

Until recently, there were two different ways to use a GPU. The earlier method was called ***General Purpose computing on a GPU*** (GPGPU) and evolved as a means to do general purpose computing by using the GPU's graphics API to do non-graphics tasks. Subsequently, as GPUs included more general purpose instructions in their instruction sets, it became possible to write non-graphics programs using a parallel programming language and API. This paradigm was called ***GPU computing***.

In 2007, NVIDIA released a software architecture and computational model to make it easier to write C or C++ programs that could exploit the high degree of parallelism in the GPU. They called this model, ***Compute Unified Device Architecture***, or ***CUDA***, for short. CUDA consists of a software library and a compiler that maps the enhanced C/C++ code into instructions for the GPU and CPU. It essentially allows the programmer to write highly parallel C/C++ programs by using additional data types and functions provided by the library. In fact, CUDA can be used to write parallel programs for multiple-core CPUs as well.

There are three general concepts that define CUDA:

- A hierarchy of computational units
- A hierarchy of shared memories
- A hierarchy of barrier synchronization

The computational hierarchy consists of ***threads*** at the lowest level, to ***thread blocks***, which are groups of threads, to ***grids***, which are groups of blocks. Each level has an associated memory that maps naturally to the physical memories inside the GPU. Similarly, there are methods of barrier synchronization that can be

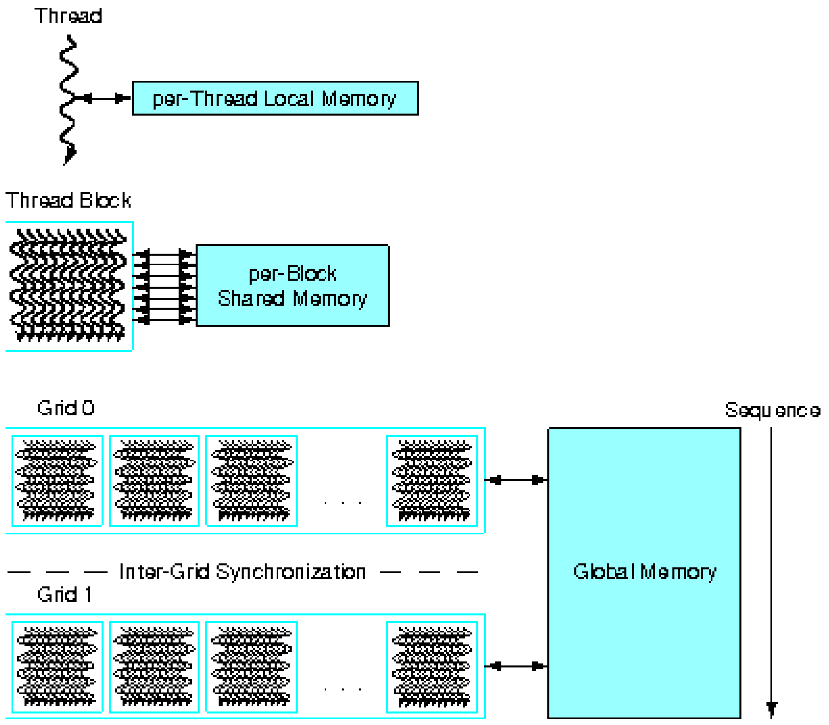

Fig. 8: Hierarchy of computational units and memories.

used with threads or with thread blocks. Figure 8 shows the relationship between computational units and their associated memories.

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 8. Each thread has private local memory, called ***per-thread memory***. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. This is called ***per-block shared memory*** in the figure. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the ***constant*** and ***texture*** memory spaces. (In the Kepler models, each SMX has a separate 48 KB read-only data cache accessible to all threads in a thread block.) We will see later that the global, constant, and texture memory spaces are optimized for different memory usages. The global, constant, and texture memory spaces are persistent across invocations of threads within the same application. In other words, these three memory spaces retain their data as threads are created and destroyed. A large part of the challenge of GPU computing is understanding the capacities and methods of access of each different type of memory in order to maximize performance.

A program consists of different kinds of data and functions. Certain variables and data are local to a single thread, others to thread blocks, and others, to the entire application. Similarly, special functions called ***kernels*** are functions that are executed in parallel by all threads. An example of a kernel is the following, which adds two arrays `A` and `B` of size N and stores the result into array `C`:

```
__global__
void VecAdd(float* A, float* B, float* C )
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

The qualifier `__global__` is a CUDA extension to C that identifies the function as a kernel function, which means that every thread executes it. Each thread is given a unique integer identifier, which in this case is stored in the variable `threadIdx.x`. To call this function from the main program, one would use the syntax

```
int main()
{
    ...
    // Kernel invocation with one block of N threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N);
    VecAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

The triple-angle brackets <<<...>>> identify the ***execution configuration*** of the kernel, which indicates how many blocks per grid (up to three dimensions) and how many threads per block (up to three dimensions). In this example, the grid consists of one block because `numBlocks` = 1, and the block is a set of N threads. `threadsPerBlock` is declared as a `dim3`, which is a 3-dimensional structure having `x`, `y`, and `z` members; the initialization sets `x`=`N` and `y`=`1` and `z`=`1` by default.

This is the flavor of computing in CUDA, but there is much more to be said about it later. For now, the important observation is that the CUDA programming model requires an underlying architecture that can execute many threads extremely quickly, and be able to switch among groups of threads as well. It also has to provide the different memory spaces in an efficient way. This leads us to explore the multithreading capabilities of the GPU.

1.7 The GPU's Multithreaded Multiprocessor

The GPU processor's multithreading is designed to achieve several goals:

- *To hide the latency of memory loads and texture fetches from DRAM and shared block memories.*

  Memory accesses can take hundreds of processor cycles. Multithreading allows the processor to switch to another thread while one thread is waiting for a load or texture fetch to complete. The extremely high degree of multithreading can keep many cores busy even though many threads might be stalled waiting for memory loads, because if there are enough active threads, then the probability that there are threads to keep all cores busy will be high.

- *To support fine-grained parallel graphics shader programming models and parallel computing models.*

  Graphics shader programs typically execute many different stages dynamically, from vertex shading to pixel shading. Because of this, the streaming multiprocessors are designed to execute different thread programs concurrently.

- *To virtualize the physical processors as threads and thread blocks in order to make them highly scalable.*

  Furthermore, each thread can have its own private registers, private memory, program counter, and thread execution state, and can execute its own independent code sequence. To make all of this possible, the GPU multiprocessor is hardware multithreaded, managing hundreds of threads without scheduling overhead. Threads within a thread block can synchronize with each other using a barrier synchronization instruction (like the one we saw in Chapter 7 notes).

- *To simplify the parallel programming model so that the programmer only has to write serial kernel functions.*

  All of the concurrency in the CUDA extensions to C and C++ takes place in kernel functions. This has the potential to simplify the logic of many data parallel programs.

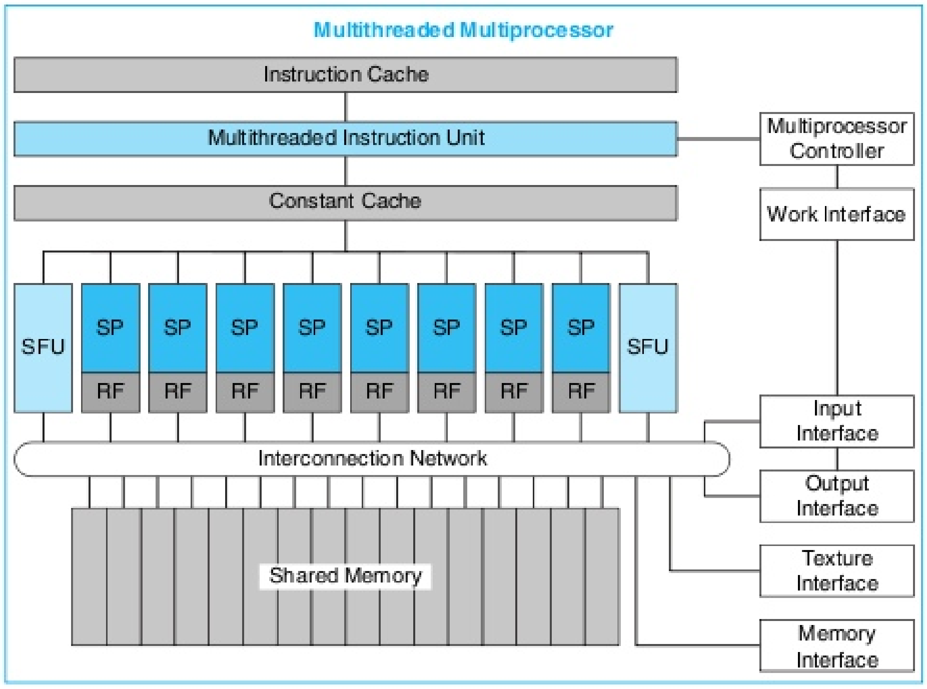

Fig. 9: A single multithreaded multiprocessor

1.7.1 The Multiprocessor Architecture

Figure 9 shows the architecture of a generic streaming multiprocessor. A streaming multiprocessor has a fixed number of cores that varies from one model to another. There are eight cores in the G80 and GT200 generation GPUs and 32 in the Fermi models. There are 192 cores in the Kepler SMX models. Each SM has a shared memory, indicated in CUDA programs by the `__shared__` qualifier in variable declarations. Shared variables are accessible to all threads in a thread block.

Each SP core has scalar integer and floating point arithmetic units that execute most of its instructions. It is hardware multithreaded and can support anywhere from 32 to 96 threads, depending on the model. The SP core is also pipelined, and can run several threads concurrently. The program declares how many of the SP's 1024 registers each thread needs. If each thread uses few registers, then more threads can run concurrently. If each thread uses many registers, then fewer threads can run at a time. For example, a pixel shader program usually uses no more than 16 registers per thread, so the 1024 registers can be allocated to 64 threads, which implies that each SP core can run 64 pixel shader threads concurrently.

The SM uses a model something like SIMD, called ***Single Instruction Multiple Thread*** or ***SIMT*** for short. It creates groups of threads called ***warps***[2]. A ***warp*** is a group of 32 threads scheduled on a single SM together. A single SM will generally execute multiple warps, interleaving their executions to hide stalls, as shown in Figure 10. A thread block is never smaller than a warp, though thread blocks can consist of multiple warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

In the Tesla, warps are scheduled onto the cores in units of four clock cycles; a warp of 32 threads is distributed among 8 cores with four threads per core. Over four clock cycles, each of the four threads executes an instruction, so that after four clock cycles, 32 threads have executed one instruction each. In the Fermi architecture, each SM has *two warp schedulers* and two instruction dispatch units, allowing two warps

[2] The term warp originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp. A quarter-warp is either the first, second, third, or fourth quarter of a warp.

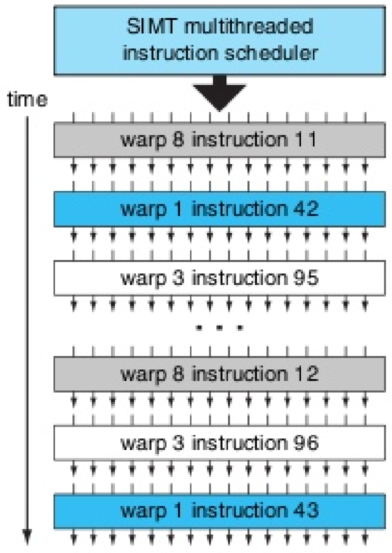

Fig. 10: SIMT multithreaded warp scheduling

to be issued and executed concurrently. Fermi's ***dual warp scheduler*** selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs. Running two warps in parallel increases the ability to hide stalls due to memory accesses. In the Kepler architecture, each SMX has *four warp schedulers* and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's ***quad warp scheduler*** selects four warps, and two independent instructions per warp can be dispatched each cycle. Figure 11 depicts one of the four warp schedulers' actions over time. Unlike Fermi, which did not permit double precision instructions to be paired with other instructions, the Kepler GK110 allows double precision instructions to be paired with other instructions.

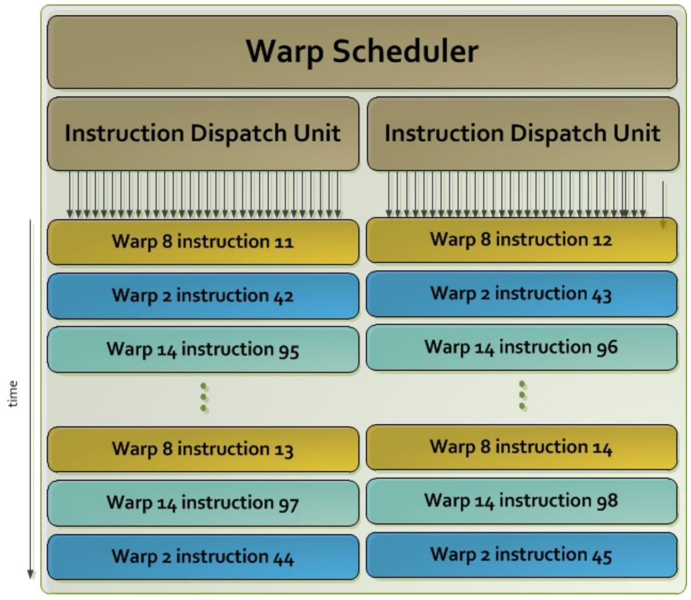

Fig. 11: A single Kepler dual-dispatch warp scheduler.

The SM tries to identify which threads can execute simultaneously and schedules them dynamically. The processors can handle data hazards and conditional instructions by allowing threads to follow different paths.

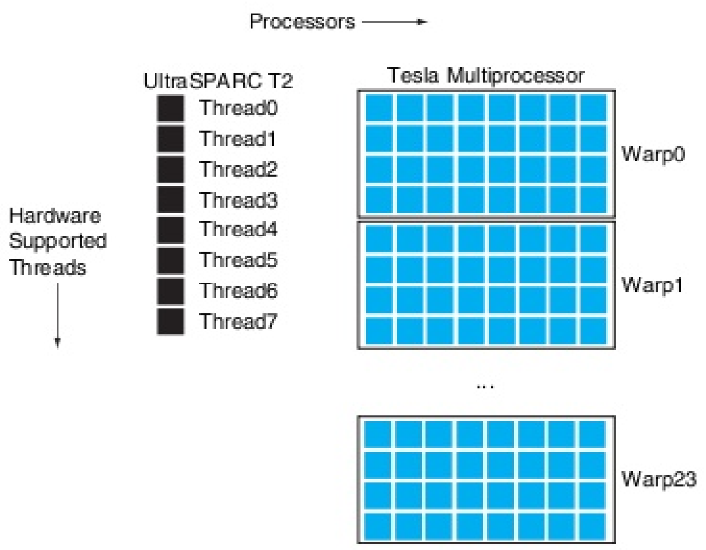

Fig. 12: Comparison of Sun UltraSPARC T2 to a single Tesla, 8-core SM

Because some threads are executing different paths than others, certain threads will have to be inactive at times (otherwise they would be executing the wrong instructions). Thus, when branch conditions evaluated on different data cause different paths to be followed, the throughput is diminished. When the threads rejoin, they start executing in unison again. If the paths through the code on different branches are the same length, the performance loss is not as great as when they are of unequal lengths.

The Tesla uses fine-grained multithreading to schedule 24 warps over time, which run in blocks of four cycles each. The Tesla will not switch threads more frequently than every two clock cycles. Figure 12 illustrates how warps are scheduled on a single Tesla SM. As noted above, each warp is distributed across eight cores. Since there are 32 threads, at its best, it will take 4 cycles to execute all threads of the warp. It will take longer if there are many conditional branches and consequent inactive threads.

The SM's SIMT multithreaded instruction unit picks a warp that is ready to execute its next instruction (because it has enough active threads) and issues that instruction to the active threads in that warp. There may be warps of different types being executed concurrently in a single SM; a warp of pixels may execute concurrently with a warp of vertices.

1.7.2 SIMT Warp Execution and Divergence

A warp executes one common instruction at a time; full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

1.8 GPUs in Perspective

GPUs do not fit into any classification scheme. Parts of it are MIMD and parts of it are SIMD. The NVIDIA GeForce 8800 GTX, for example has 16 identical Tesla processors, each of which is a sort of multiprocessor.

Thus, from a large scale view, it is MIMD. Each Tesla processor consists of eight streaming processors, with a SIMD architecture. Each streaming processor executes the same instruction on different data. Within the processor there is a shared memory that each streaming processor can access.

Although the hardware is SIMD, the Tesla programmer interface creates the illusion that the multiprocessor is MIMD. It achieves this by making certain threads inactive when they are not supposed to execute a given instruction, and by high level parallel fine-grained multithreading. At its best, all 32 threads in a warp are busy all of the time. The hardware is more like SIMT – single instruction multiple thread, than SIMD. But if the programmer does not write the code carefully, the machine will not take advantage of the maximum parallelism possible.

2 GPU Programming and CUDA

There are really two fundamental issues in programming a GPU – correctness and performance. Of course it is imperative that the programs are correct, but because the reason that one chooses to write a program to run on a multiprocessor is to decrease running time and/or increase problem size, performance is almost as important. Thus, while a programmer can generally ignore the design of the SIMT architecture and ignore how warps are scheduled and executed in terms of correctness, he or she can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses, and take advantage of knowledge of how global and shared memory are accessed.

Another issue is a practical one; how does one actually write general purpose programs that can be run on a GPU? There are different approaches to this problem, but the easiest solution is to use NVIDIA's CUDA extension to the C/C++ language. These notes present an overview of CUDA Version 4.0. They are not intended as a reference manual nor as a technical guide. Most of the material comes from the NVIDIA CUDA C Programming Guide, Version 4.0.

CUDA extends C by adding constants, types, and functions that expose the capabilities of a GPU. As noted earlier, it consists primarily of three key abstractions:

- a hierarchy of thread groups,
- shared memories, and
- barrier synchronization,

that allow the programmer to write programs that have fine-grained data parallelism and thread parallelism, as well as coarse-grained data parallelism and task parallelism.

For the purpose of writing correct programs, it is enough to learn the syntax and semantics of a relatively small subset of CUDA, but for writing programs with optimal performance, it is important to understand the underlying execution model and the memory model. We start with an overview of the memory model and then look at some of the details of CUDA.

2.1 The Different Types of Memory Accesses

There are various levels of memory that a thread can access explicitly - thread private, shared, constant, texture, and global. An instruction that accesses addressable memory might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. How the distribution affects the instruction throughput this way is specific to each type of memory and described in the following sections. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is. For shared memory, certain patterns result in higher bandwidth.

2.1.1 Global Memory Accesses

Global memory is off-chip and generally has access times that are on the order of a hundred times longer than the on-chip shared memory and caches. Therefore, one wants to minimize access to global memory. Unfortunately, the on-chip shared memory is often not large enough to store all of the data needed by the threads executing within the thread blocks. The problem facing the programmer is understanding the most efficient ways to move data between global memory and shared or thread private memory.

Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e. its address is a multiple of that size). If the read (or write) requests from multiple threads can be made to addresses in global memory that satisfy these constraints, then the reads (or writes) are coalesced into a single instruction of higher bandwidth. This will be illustrated by example below.

2.1.2 Shared Memory Accesses

Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing shared memory is fast as long as there are no ***bank conflicts*** between the threads, as detailed below.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called ***banks***, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts.

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in shared memory in order to schedule the memory requests so as to minimize bank conflicts.

For devices of compute-capability 1.x, shared memory has 16 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. Each bank has a bandwidth of 32 bits per two clock cycles. A shared memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. As a consequence, there can be no bank conflict between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.[3] Again, this is best illustrated with an example.

2.2 Host and Device: What Runs Where

A program runs as two separate pieces: the ***host code*** and the ***device code***. Host code is code that runs on the host CPU. All references to the concept of the host are references to the CPU that spawned the GPU computation. In the Tesla and Fermi models, there could be only one host using the GPU at a time, but the Kepler allows multiple hosts to share the GPU. Device code is code that runs on the GPU.

2.3 CUDA Extensions to C Types

CUDA adds vector types to the standard set of C elementary types. To be precise, for each of the types

[3] For devices of compute-capability 2.x, shared memory has 32 banks. Therefore, unlike for devices of lower compute capability, there may be bank conflicts between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

```
char, uchar, int, uint, short, ushort, long, ulong, longlong, ulonglong, float
```

if *X* is one of these types, then there are vector types *X1*, *X2*, *X3*, and *X4*. For example, there are types `int1`, `int2`, `int3`, and `int4`. The term *vector type* is a misnomer – these are not vectors, but structures. The 1^{st}, 2^{nd}, 3^{rd}, and 4^{th} components are accessible through the members x, y, z, and w, respectively. For example, if we declare

```
uint3 point;
```

then `point.x`, `point.y`, and `point.z` are the members of `point`. There are also `double1` and `double2`, but not `double3` or `double4`.

The type `dim3` is a an extension of `uint3`. It is used to specify the dimensions of things such as thread blocks and grids. Unlike `uint3` though, when an object of type `dim3` is declared , its constructor initializes all uninitialized components to the value 1. For example,

```
        dim3  block(64, 64);
```

specifies that `block` is 64 by 64 by 1, because the `z` member is set to 1.

2.4 Variable Qualifiers: Where Things Are Located

The following rules can be used to determine in which memory a variable resides.

1. An automatic variable declared in device code without any of the qualifiers `__device__`, `__shared__` and `__constant__` usually resides in a register.
2. The `__device__` qualifier declares a variable that resides on the device (meaning global memory), has the lifetime of an application, and is accessible from all the threads within the grid and from the host through the runtime library (with specific functions designed to allow the host program to access the GPU's device memory.)
3. The `__shared__` qualifier, optionally used together with `__device__`, declares a variable that it resides in the shared memory space of a thread block, has the lifetime of the block, and is only accessible from all the threads within the block.
4. The `__constant__` qualifier, optionally used together with `__device__`, declares a variable that resides in constant memory space, has the lifetime of an application, and is accessible from all the threads within the grid and from the host through the runtime library (with specific functions.)

2.5 Kernels

A ***kernel*** is a C function that is executed in parallel by more than one CUDA thread, as opposed to only once like an ordinary C function. A kernel is defined using the `__global__` declaration specifier. For example,

```
// kernel definition
__global__
void VecAdd(float* A, float* B, float* C )
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

specifies to the compiler that `VecAdd()` is a kernel. The number of threads that execute the kernel is determined by how the function is called. CUDA has a special *execution configuration* syntax for this purpose. Any call to a `__global__` function must specify the execution configuration for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device. The syntax of the call is

```
function_name<<< Dg, Db, Ns, S >>>(argument_list);
```

where

- *Dg* is of type `int` or `dim3` and specifies the dimension and size of the grid, such that *Dg.x* * *Dg.y* * *Dg.z* equals the number of blocks being launched[4];
- *Db* is of type `int` or `dim3` and specifies the dimension of the block. *Db.x* * *Db.y* * *Db.z* equals the number of threads being launched in each block.
- *Ns* is of type `size_t` and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array. Ns is an optional argument which defaults to 0.
- *S* denotes a `cudastream_t`, which we will ignore here. It is an optional argument that defaults to 0.

We could invoke the above kernel in the main program as follows:

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

In this case, *Dg*=1 and *Db*=N, so there is there is a single block containing N threads.

Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable. The `threadIdX` variable is of type `uint3`, so (`threadIdx.x, threadIdx.y, threadIdx.z`) are the coordinates of the thread within the block. Coordinates are Cartesian, not matrix. In other words, the `x` coordinate is the column position and the `y` coordinate is the row position.

Each block is given a unique block ID that is accessible within the kernel through the built-in `blockIdx` variable. The `blockIdX` variable is of type `uint3`, so (`blockIdx.x, blockIdx.y, blockIdx.z`) are the coordinates of the block within the grid.

The dimensions of the grid are specified with up to three dimensions and these dimensions are accessible to each thread in the kernel through the built-in `blockDim` variable, which is of type `dim3`.

To illustrate, suppose we want to process a matrix M that has 32 columns and 18 rows by dividing it into a grid that has 4 columns and 3 rows of blocks, as shown in Figure 13. Each block will consist of 6 rows and 8 columns of unique threads. The main program would call our kernel using the execution configuration

```
dim3  gridSize(4,3);
dim3  blockSize(8,6);
float A[32][48];

ProcessMatrix<<<gridSize, blockSize>>>(A);
```

[4] *Dg.z* must be equal to 1 for devices of compute capability 1.x.

and the kernel function might be something like this:

```
// kernel definition
__global__
void ProcessMat(float** A )
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    // process A[row][col]
}
```

Figure 13 shows how matrix element M[9][17] is being processed by the thread with id (1,3) in the block with id (2,1).

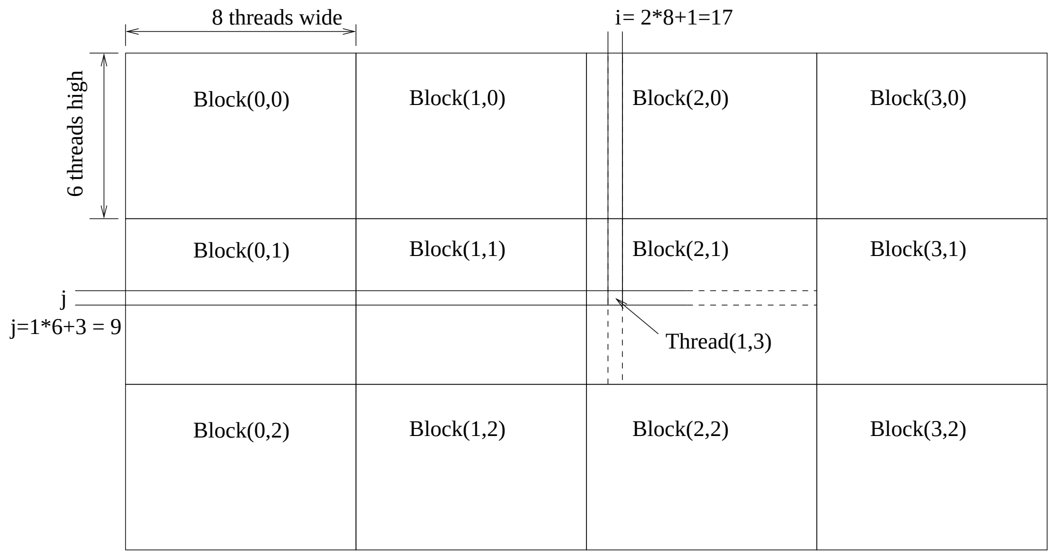

Fig. 13: Decomposition of a matrix into a grid of thread blocks. In this example, the matrix is 18 rows by 32 columns. Each thread block has 8 columns and 6 rows of threads. A thread id within a block is (x, y), $0 \le x \le 7$, $0 \le y \le 5$. The figure shows how a thread with id (1,3) within the block with id (2,1) is associated with the matrix entry (17,9). Matrix elements are referred to using Cartesian coordinates – column first, then row.

2.6 Matrix Multiplication Example

We illustrate CUDA programming using matrix multiplication as an example. Matrix multiplication is so important because it arises in many different contexts and can be time-consuming. For those who have not yet had a course in matrix algebra, we begin with what it means to find the product of two matrices. To begin, when we write that a matrix A is $r \times s$ we mean that it has r rows and s columns.

Matrix multiplication is not commutative – AB is not the same as BA in general, so when we say that we are multiplying a matrix A on the right by matrix B, we mean the product AB. We can multiply a matrix A that is $r \times s$ on the right by a matrix B only if it has s rows. In other words, B must be of dimensions $s \times t$ for some t. The formal definition is that if A is an $r \times s$ matrix and B is an $s \times t$ matrix then the product $C{=}AB$ is the $r \times t$ matrix whose entries are defined as

$$C_{i,j} = \sum_{k=1}^{s} A_{i,k} \cdot B_{k,j} \quad 1 \le i \le r, \ 1 \le j \le t$$

In other words, $C_{i,j}$ is the "dot" product (inner product) of the i^{th}row of A and the j^{th} column of B. The dot product is only defined if these are the same length, which is why the width of A must match the height of B. Below is an example.

$$\begin{pmatrix} 1 & 3 \\ -1 & 2 \\ -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 13 & 11 & 9 & 7 \\ 7 & 4 & 1 & -2 \\ 2 & -1 & -4 & -7 \end{pmatrix}$$

Make sure you understand this example before continuing.

2.6.1 Sequential Algorithm to Multiply Two Matrices

First, to make sure you understand the problem, following is an ordinary sequential C++ function that multiplies two matrices A and B of the given dimensions, storing the result in C:

```
void multiply( float** A, float** B, float** C,
               int Aheight, int Awidth, int Bwidth)
// precondition: Awidth == Bheight
{
    for ( int i = 0; i < Aheight; i++ ) {
        for ( int j = 0; j < Bwidth; j++ ) {
            C[i][j] = 0;
            for ( int k = 0; k < Awidth; k++ ) {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

The matrices are passed as dynamic arrays because otherwise one dimension of each matrix would have to be fixed size, making it a rather useless function. Client code that allocated the matrices and called this function might look like the following:

```
float** A = new float*[Aheight];
for ( i = 0; i < Aheight; i++ ) {
    A[i] = new float[Awidth];
}

float** B = new float*[Bheight];
for ( i = 0; i < Bheight; i++ ) {
    B[i] = new float[Bwidth];
}

float** C = new float*[Aheight];
for ( i = 0; i < Aheight; i++ ) {
    C[i] = new float[Bwidth];
}

multiply(A,B,C,Aheight,Awidth,Bwidth);
```

Figure 14 illustrates how a single entry of the product is obtained. This matrix multiplication algorithm requires $O(n^3)$ multiplications and additions. Now we see how it can be done using the GPU.

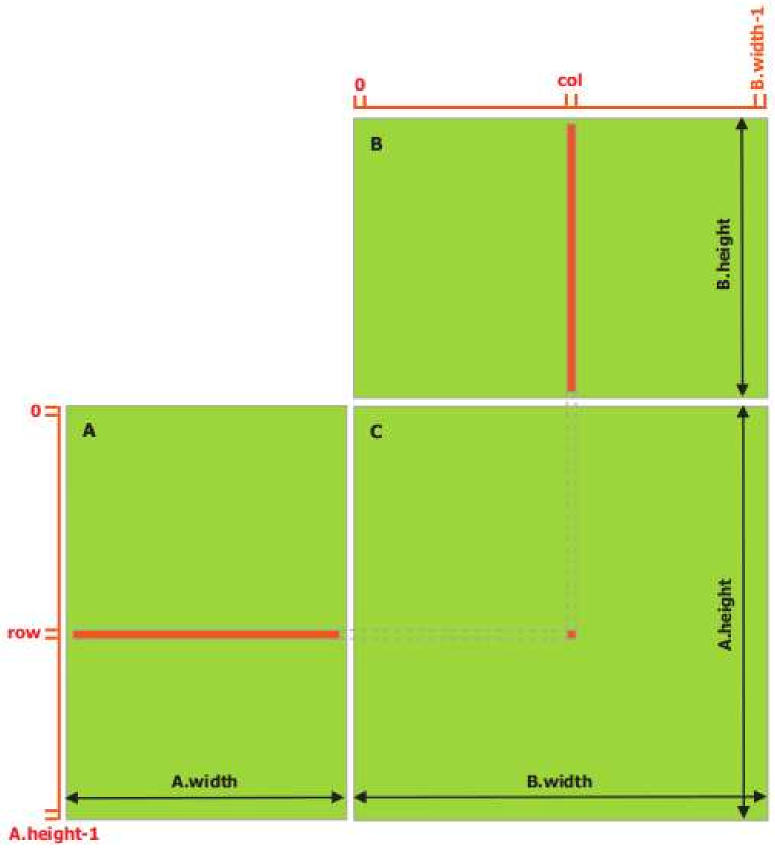

Fig. 14: Matrix Multiplication (without shared memory)

2.6.2 Matrix Multiplication the Simple Way

We start with a method of multiplying matrices that is easy to understand but which does not take advantage of the different access times of shared and global memory in the GPU.The code uses the following definition of a `Matrix` structure:

```
// The matrix is stored in a linear array. Conceptually
// M[row][col] = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

In C/C++ and in CUDA on the GPU, two-dimensional matrices are stored in row-major order. This means that row 0 is stored first, followed by row 1, then row 2, and so on. We can represent a two-dimensional matrix in one dimension using the above representation.

The listing below is not a main program, but a function that can utilize the resources of the GPU to perform matrix multiplication. It computes the matrix product $C = AB$, where A and B are matrices of type float.

The macro

```
#define BLOCK_SIZE 16
```

declares the thread block size, which will be a square block of 256 threads.

The Algorithm This code is an implementation of matrix multiplication that does not take advantage of the shared memory within each SM (streaming multiprocessor). There will be a thread for each cell of the C result matrix. Let A be an M by N matrix and B, a N by R matrix. We assume that M,N, and R are divisible by 16, the block size for this implementation.

The A and B matrices are copied from host memory to device memory. There, a kernel is run on a grid of blocks of threads, each of which computes a single cell of C. To compute a single cell `C[i][j]` of C, it has to read the i^{th} row of A and the j^{th} column of B, each of size N. Each cell does this. Many cells read the i^{th} row of A and many read the j^{th} row of B, but only one does both of these uniquely.

To be clear, each row of A is read R times (for each column of B) and each column of B is read M times (for each row of A). Therefore, A is read R times from global memory and B is read M times. The basic steps are

1. Allocate memory on the device for matrix A, and copy A onto the device's global memory.
2. Do the same for matrix B.
3. Allocate space on the device for C, the matrix product.
4. Compute the shape of the grid and the number of blocks needed to cover the product matrix C.
5. Launch the kernel with the grid and blocks.
6. When the kernel is done, copy matrix C from the device memory to host memory.
7. Free the device memory used by A, B, and C.

The listing follows.

Listing 1: Matrix multiplication without shared memory

```
/*
  CUDA-based MATRIX MULTIPLICATION Without Using Shared Memory

  Functions from CUDA used here include:
    cudaError_t cudaMalloc(void** devPtr, size_t count )
        This allocates count bytes of linear memory on the GPU and returns
        in *devPtr a pointer to the allocated memory. The allocated memory
        is suitably aligned for any kind of variable. The memory is not zeroed.
        cudaMalloc()
        returns cudaErrorMemoryAllocation in case of failure.
        cudaSuccess  on success.

    cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                           enum cudaMemcpyKind kind)
        This copies count bytes from the memory area pointed to by src to the
         memory area pointed to by dst, where kind is one of
            cudaMemcpyHostToHost,
            cudaMemcpyHostToDevice,
            cudaMemcpyDeviceToHost, or
            cudaMemcpyDeviceToDevice,
        and specifies the direction of the copy. The memory areas may not overlap.
        Calling cudaMemcpy() with dst and src pointers that do not match the
        direction of the copy results in an undefined behavior.
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```
#include <cuda.h>
#include <cuda_runtime_api.h>
#include "matrixmult.h"

#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication  kernel (in separate listing)
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

/*
    Matrix multiplication - Host code
    Matrix dimensions are assumed to be multiples of BLOCK_SIZE
*/
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    size_t size;

    // Declare the matrix that will reside on the device (the GPU)
    Matrix d_A;
    d_A.width  = A.width;
    d_A.height = A.height;

    /* Calculate its size in bytes */
    size = A.width * A.height * sizeof(float);

    /* Next allocate space for d_A on the device. This uses the function
       cudaMalloc(). It fills d_A.elements with the location of the start
       of storage for d_A.  */
    cudaMalloc(&d_A.elements, size);

    /* Next, copy the array from the host memory to the device memory
       using cudaMemCpy(). */
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

    /* Repeat the above steps for the matrix B */
    Matrix d_B;
    d_B.width = B.width;
    d_B.height = B.height;

    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

    /* Repeat almost all of the steps for the C matrix. We do not copy it
       from host memory because it gets computed on the device and sent back
       to the host.
    */
    Matrix d_C;
    d_C.width = C.width;
    d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    /* It is time to invoke the kernel on the GPU. The kernel is run by
       several streaming multiprocessors. The C result matrix is decomposed
       into square blocks of size BLOCK_SIZE by BLOCK_SIZE. Each of these
       is computed by a block of threads. There will be a grid of such blocks.
       If the B matrix is R cells wide and the A matrix is M cells high, then
       C is M by R. Assuming R and M are multiples of BLOCK_SIZE, the grid
       is of width R/BLOCK_SIZE and of height M/BLOCK_SIZE. The next two
       instructions define the block and grid dimensions.
    */
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);

    /* Now the kernel is invoked on this grid of blocks of threads */
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    /* When the kernel completes, we read C from device memory back to
       host memory.
    */
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    /*
      cudaError_t cudaFree(void* devPtr)
      This frees the memory space pointed to by devPtr, which must have been
      returned by a previous call to cudaMalloc() or cudaMallocPitch().
      Otherwise, or if cudaFree(devPtr) has already been called before,
      an error is returned. If devPtr is 0, no operation is performed.
    */
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

The matrix multiplication kernel is in the listing below:

Listing 2: Matrix multiplication kernel.

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    /* Each thread computes one element of C
       by accumulating results into Cvalue
    */
    float Cvalue = 0;
    int k;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for ( k = 0; k < A.width; k++)
        Cvalue += A.elements[row * A.width + k] *
                  B.elements[k * B.width + col];

    C.elements[row * C.width + col] = Cvalue;
}
```

The if statement terminates the thread if its row or column place it outside the bounds of the product matrix. This will happen only in those blocks that overhang either the right or bottom side of the matrix. The next three lines loop over the entries of the row of A and the column of B (these have the same size) needed to compute the (row, col)-entry of the product, and the sum of these products is accumulated in the `Cvalue` variable. The last line of the kernel copies the sum of the products into the appropriate element of the product matrix C, in the device's global memory.

2.6.3 Matrix Multiplication Using Shared Memory

The problem with the simplified algorithm we just explored is that it makes many accesses to global memory. In the loop in the kernel code, each thread loads (2*A.width) elements, two for each iteration through the loop, one from matrix A and one from matrix B. Since accesses to global memory are relatively slow, this slows down the kernel code, leaving many threads idle for hundreds of clock cycles.

One way to reduce the number of accesses to global memory is to have the threads load portions of matrices A and B into shared memory, where they can access them much more quickly. The problem is that shared memory is not large enough to store two large matrices. Devices of compute capability 1.x have 16 KB of shared memory per multiprocessor, and devices of compute capability 2.x have 48 KB.

So instead, portions of A and B are loaded into shared memory as needed, and utilized as efficiently as possible while they are loaded. Figure 15 shows how the matrix product can be computed in a block-structured way. Matrix A is shown on the left and matrix B is shown at the top, with matrix C, their product, on the bottom-right. Each element of C is the product of the row to its left in A, and the column above it in B. The matrices A and B are partitioned into 16 by 16 submatrices (`BLOCK_SIZE`=16.)

Each thread block is responsible for computing one square sub-matrix (`Csub` in the code) of the product matrix C, and each thread will be responsible for computing one element of the product matrix C. (The yellow square in matrix C represents the submatrix computed by a thread block, whereas the red box inside the yellow square represents a single entry in C, what is computed by a single thread within that block.) The thread responsible for this entry computes its value by computing the dot product of the red row of A and the red column of B. Unfortunately, it is not so simple as this, because it must do this in pieces, as not all of its input data will be in shared memory at the same time.

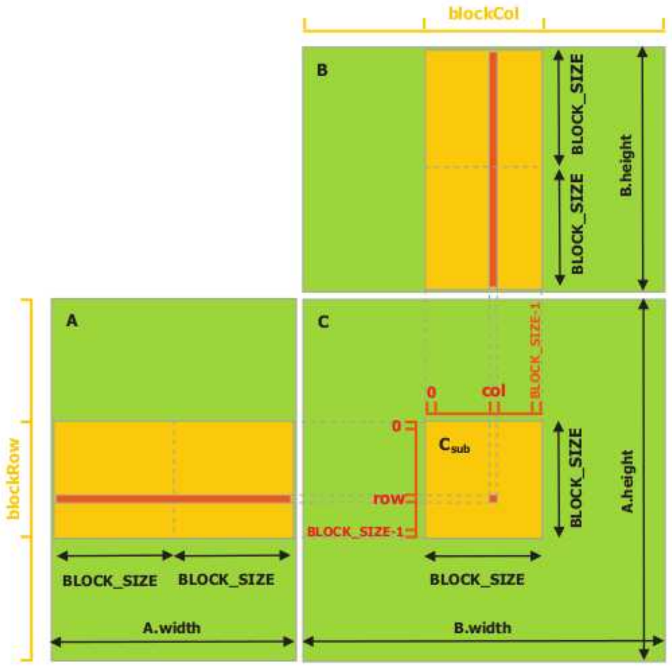

Fig. 15: Matrix Multiplication (using shared memory)

`Csub` is equal to the product of two rectangular matrices: the sub-matrix of A of dimension `A_width` x `BLOCK_SIZE` that has the same row indices as `Csub`, and the submatrix of B of dimension `BLOCK_SIZE` x `A_width` that has the same column indices as `Csub`. These are the yellow rectangular regions within the A and B matrices in Figure 15. These regions will not all be in shared memory together. But notice that the red row and red column pass through the same number of submatrices, since they are of equal length. This leads to the idea: if we load the left-most of those submatrices of matrix A into shared memory, and the top-most of those submatrices of matrix B into shared memory, then we can compute the first `BLOCK_SIZE` products of the dot-product entirely from the shared memory. Every thread in the thread block can do this for its own little red square from the figure, and because those submatrices are in shared memory, making this very fast.

When this is done, we no longer need the left-most block of A and the top-most block of B. We load from global memory to shared memory the block of A to the right of the previous one, and the block of B below the previous one and repeat the above step. Each thread computes the dot product of the next `BLOCK_SIZE` entries. This gets added to the running total that the thread is maintaining for its *(i,j)* entry. This process continues until the entire row of A has been multiplied by the entire column of B. When this is finished, the resulting square submatrix is written back to global memory.

To make the copying from global to shared memory efficient, each thread is responsible for copying a single element from each of the A and B matrices. The copying is done in such a way to maximize the memory bandwidth, which will be explained within the listing below.

Listing 3: Matrix multiplication using shared memory.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
// The stride of a matrix is the number of bytes from the start of a row to
// the start of the next row. The stride is not necessarily equal to the width;
// the stride can be larger so that rows are aligned in shared memory to
// achieve good bandwidth.
typedef struct {
    int width;       // width of matrix
    int height;      // height of matrix
    int stride;      // number of bytes from start of row k to start of row k+1
    float* elements; // pointer to actual matrix data
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Msub of M that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of M
// This returns a pointer to the first element of that submatrix.
__device__ Matrix GetSubMatrix(Matrix M, int row, int col)
{
    Matrix Msub;
    Msub.width    = BLOCK_SIZE;
    Msub.height   = BLOCK_SIZE;
    Msub.stride   = M.stride;
    Msub.elements = &M.elements[M.stride * BLOCK_SIZE * row
                                + BLOCK_SIZE * col];
    return Msub;
}


// The __global__ qualifier declares a function as being a kernel.
// A kernel is a function that is executed on the device (the GPU), and
// is callable from the host (CPU).
// This is a forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);


// Host multiplication function
// Compute C = A * B
// height_A is the height of A
// width_A is the width of A
// width_B is the width of B
void Multiply ( const float  *A,
                const float  *B,
                int          height_A,
                int         width_A,
                int         width_B,
                float       *C)
{
    int size;

    /*
       Copy matrices A and B to the device memory
       To copy requires using cudaMemCpy, which can copy either from host
       to device, from device to host, or from device to device.
    */
    float* A_on_device;
    size = height_A * width_A * sizeof(float);
    cudaMalloc((void**)&A_on_device, size);
    cudaMemcpy(A_on_device, A, size, cudaMemcpyHostToDevice);


    float* B_on_device;
    size = width_A * width_B * sizeof(float);
    cudaMalloc((void**)&B_on_device, size);
    cudaMemcpy(B_on_device, B, size, cudaMemcpyHostToDevice);

    // Allocate matrix C on the device
    float* C_on_device;
    size = height_A * width_B * sizeof(float);
    cudaMalloc((void**)&C_on_device, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    // The dim3 declaration is used here. This specifies that dimBlock
    // is BLOCK_SIZE x BLOCK_SIZE x 1
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    // width_B/dimBlock.x is the length of a row of B divided by the horizontal
    // size of a block, which yields the number of blocks in the horizontal
    // dimension of the Grid.
    // Similarly,  height_A/dimBlock.y is the height of A divided by the vertical
    // size of a block, which is the number of blocks vertically in the grid.
    // These two values define the shape of the grid, i.e. the number of
    // block horizontally and vertically.
    dim3 dimGrid(width_B / dimBlock.x, height_A / dimBlock.y);

    // Launch the device computation
    Muld <<<dimGrid, dimBlock>>> (A_on_device,
                                  B_on_device,
                                  width_A,
                                  width_B,
                                  C_on_device);

    // Copy result C from the device to host memory
    cudaMemcpy(C, C_on_device, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(A_on_device);
    cudaFree(B_on_device);
    cudaFree(C_on_device);
}
```

```
   /*
      Note again the __global__ qualifier: this is a kernel function
      This also means that every thread executes this function.
      When a thread executes this function, it has a specific thread id
      and block id. The thread id is the value of threadIdx, used below,
      and the block id is stored in blockIdx, used below.
      threadIdx  and blockIdx are each of type dim3.
   */
__global__
void Muld(float* A, float* B, int width_A, int width_B, float* C)
{
    // Block index
    int block_col = blockIdx.x;
    int block_row = blockIdx.y;

    // Thread index
    int thread_col = threadIdx.x;
    int thread_row = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = width_A * BLOCK_SIZE * block_row;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + width_A - 1;

    // Step size used to iterate through the sub-matrices of A
    // The upper left corner of the next block of A is BLOCK_SIZE columns
    // from the current block's corner, so the increment is just BLOCK_SIZE.
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * block_col;

    // Step size used to iterate through the sub-matrices of B
    // The upper left corner of the next block of B is BLOCK_SIZE rows
    // below the upper left corner of the current block. Each row has
    // width_B bytes, so the increment is BLOCK_SIZE * width_B.
    int bStep = BLOCK_SIZE * width_B;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
            a <= aEnd;
            a += aStep, b += bStep) {

        // The __shared__ qualifier declares a variable that
        //  * resides in the shared memory space of a thread block,
        //  * has the lifetime of the block, and
        //  is only accessible from all the threads within the block.
        // The As and Bs matrices declared below are in the shared memory of
        // the block; As is for the sub-matrix of A, and Bs, a submatrix of B.

        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    /*
       The next step loads the matrices from global memory to shared memory;
       each thread loads one element of each matrix.

       The global memory access by all threads of a half-warp is coalesced
       into one or two memory transactions if it satisfies the following
       three conditions:
         1. Threads must access
            Either 4-byte words, resulting in one 64-byte memory transaction,
            Or 8-byte words, resulting in one 128-byte memory transaction,
            Or 16-byte words, resulting in two 128-byte memory transactions;
         2. All 16 words must lie in the same segment of size equal to the
            memory transaction size (or twice the memory transaction size
            when accessing 16-byte words);
         3. Threads must access the words in sequence: The kth thread in the
            half-warp must access the kth word.

       In these instructions, each thread is accessing a 4-byte word (float)
       and 16 threads do this simultaneously, for a total of 64-bytes in
       each transaction. The words accessed by the threads are in sequence
       so the access is coalesced and will take just two trips to global
       memory.
    */
    As[thread_row][thread_col] = A[a + width_A * thread_row + thread_col];
    Bs[thread_row][thread_col] = B[b + width_B * thread_row + thread_col];

    // Synchronize to make sure the matrices are loaded
    // __syncthreads() is a barrier synchronization call; all threads
    // wait here until every thread has made the call, at which point it
    // returns in each thread.
    __syncthreads();

    /*
       The loop below computes the inner product of the row of A_shared
       and column of B_shared assigned to this thread, As[thread_row],
       and Bs[thread_col], i.e., the sum

         As[i][0]*Bs[0][j] + As[i][1]*Bs[1][j] + ... As[i][N-1]Bs[N-1][j]


       For a device with compute capability 1.x (most devices in use today
       with CUDA capability), there are 16 banks in shared memory, each
       4-bytes wide. A float is stored in one word of a bank.

       Remember that all threads within a single warp execute the same
       instruction at a time. Therefore, when a kernel has a loop such as
           for (int k = 0; k < BLOCK_SIZE; ++k)
               Csub += As[thread_row][k] * Bs[k][thread_col];
       each thread executes the instruction in the bosy simultaneously.

       Consider that assignment statement,

           Csub += As[thread_row][k] * Bs[k][thread_col];

       for fixed k. Each of the threads in the half-warp have the same
       value of thread_row, because a warp consists of 16 threads in
       row i followed by 16 threads in row i+1, for i = 0,2,4,... 30.
```

```
           So the first half-warp accesses only row i, and the second half-warp
           accesses only row i+1.
           Therefore, for fixed k, As[thread_row][k] is the same memory location
           and will be read once and stored locally for each thread in a
           register.

           The values Bs[k][thread_col] are 16 successive 4-byte words, stored
           in banks 0, 1, 2, ..., 15 of the shared memory, and therefore there
           is no bank conflict in their accessing these banks simulataneously.
           Therefore, by Rule 1 above, this is a 64-byte memory transaction that
           is coalesced into one or two cycles.  In short
           the for-loop below maximizes the shared memory bandwidth.

        */
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[thread_row][k] * Bs[k][thread_col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = width_B * BLOCK_SIZE * block_row + BLOCK_SIZE * block_col;
    C[c + width_B * thread_row + thread_col] = Csub;
}

void initMatrix( float M[], float value, int width, int height)
{
    int i,j;

    for ( i = 0; i < height; i++ )
        for ( j = 0; j < width; j++ )
            M[j + i*width] = value;
}

void printMatrix( float M[] ,  int width, int height)
{
    int i,j;

    for ( i = 0; i < height; i++ ) {
        for ( j = 0; j < width; j++ )
            printf("%8.2f " , M[j + i*width]);
        printf("\n");
    }
}


int main(int argc, char* argv[])
{
    float *A, *B, *C;
    int width  = MATRIXSIZE;
    int height = MATRIXSIZE;

    A = (float*) calloc(width * height, sizeof(float) );
    if ( A == NULL )
        exit(1);
    B = (float*) calloc(width * height, sizeof(float) );
    if ( B == NULL )
        exit(1);

    C = (float*) calloc(width * height, sizeof(float) );
    if ( C == NULL )
        exit(1);

    initMatrix(A, 2.0, width, height);
    initMatrix(B, 3.0, width, height);
    initMatrix(C, 0.0,width, height);

    Multiply( A, B, width, width, height, C );
    printMatrix(C, width, height);
    return 0;
}
```