Chapter 1. Introduction

1.1 From Graphics Processing to General-Purpose Parallel Computing

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 1-1.

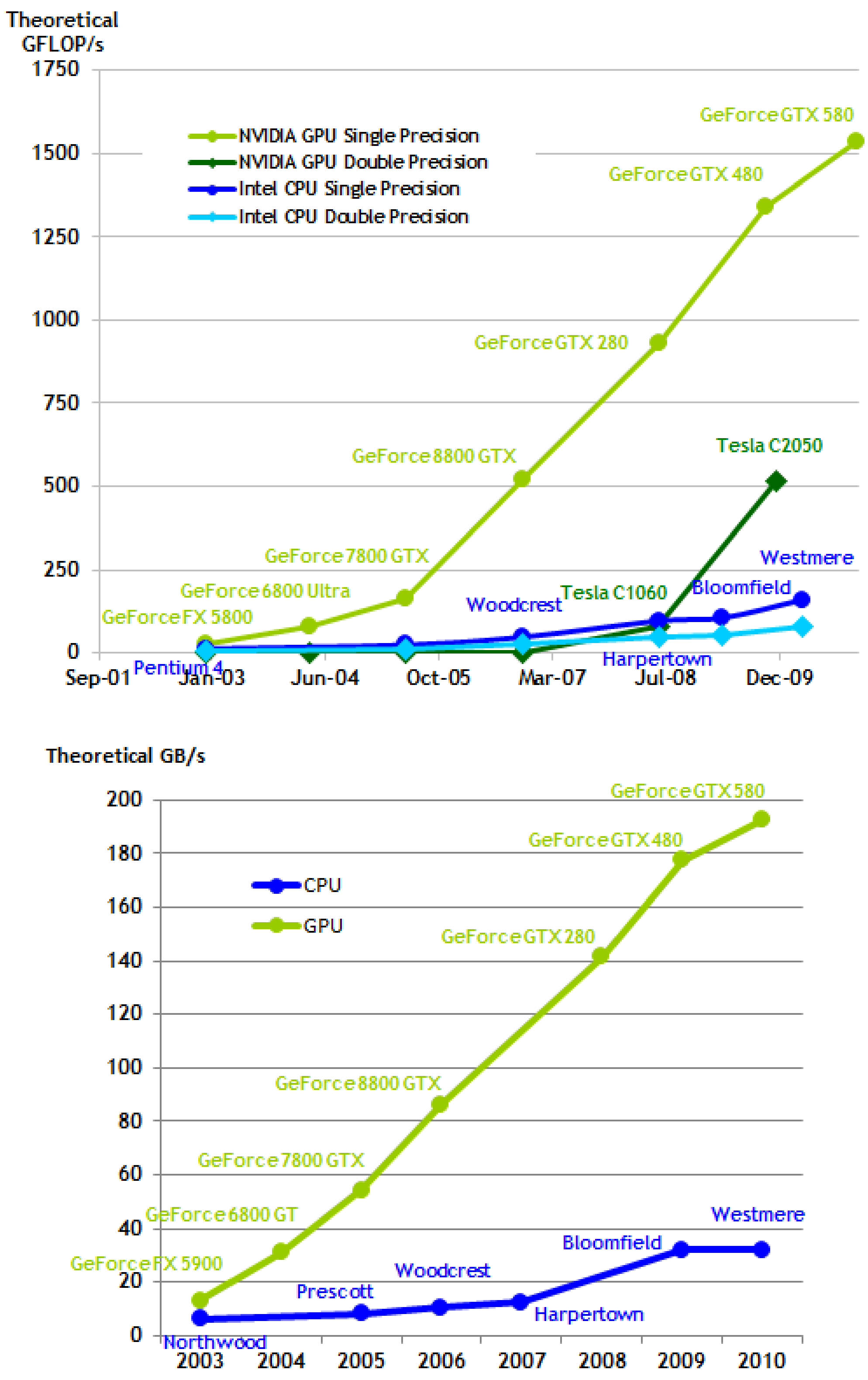


Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1-2.

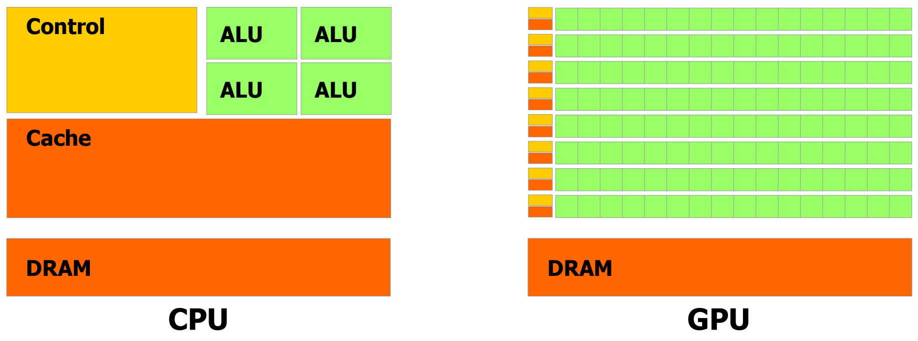


Figure 1-2. The GPU Devotes More Transistors to Data Processing

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

1.2 CUDA™: a General-Purpose Parallel Computing Architecture

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C as a high-level programming language. As illustrated by Figure 1-3, other languages or application programming interfaces are supported, such as CUDA FORTRAN, OpenCL, and DirectCompute.

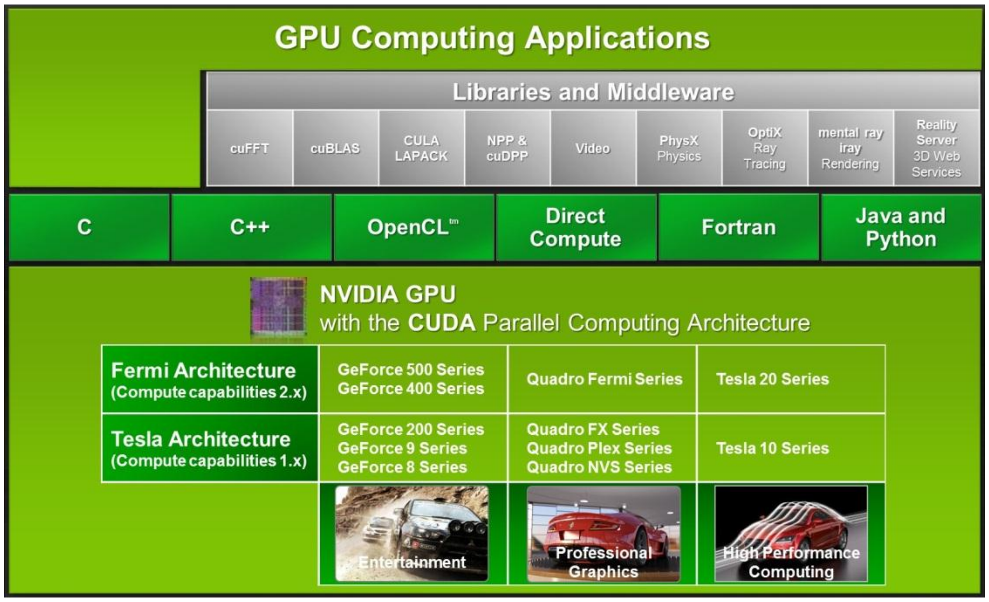


Figure 1-3. CUDA is Designed to Support Various Languages and Application Programming Interfaces

1.3 A Scalable Programming Model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

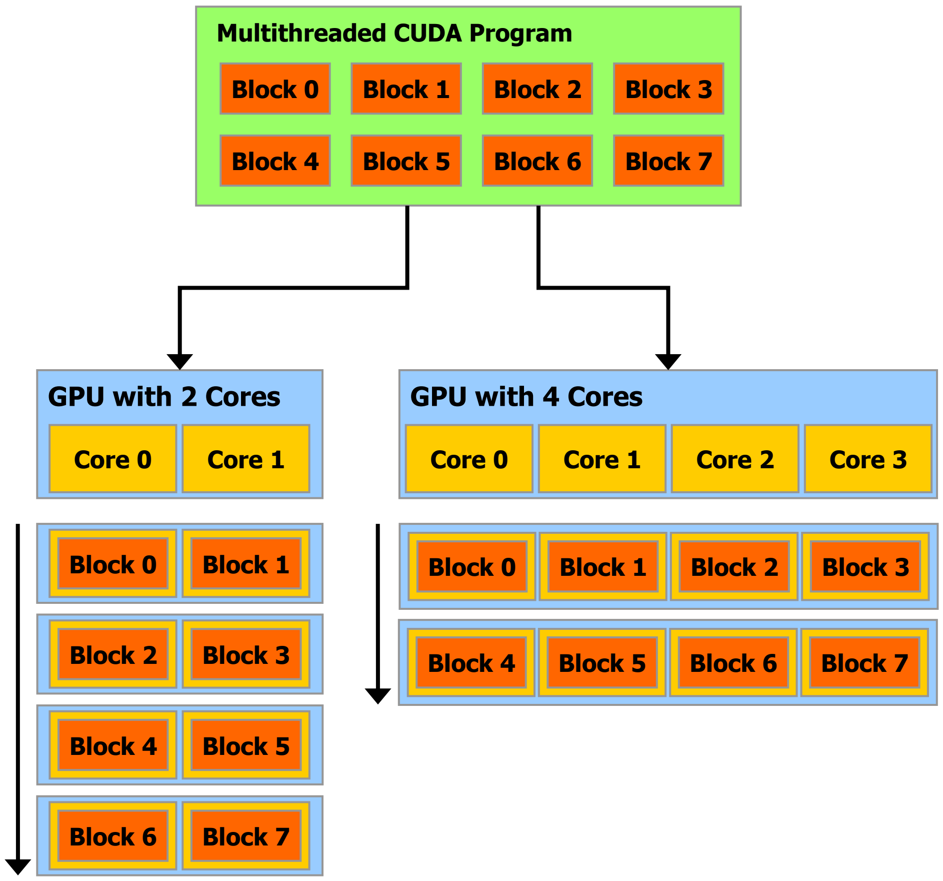
The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 1-4, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see Appendix A for a list of all CUDA-enabled GPUs).



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

1.4 Document's Structure

This document is organized into the following chapters:

- Chapter 1 is a general introduction to CUDA.
- Chapter 2 outlines the CUDA programming model.
- Chapter 3 describes the programming interface.
- Chapter 4 describes the hardware implementation.
- Chapter 5 gives some guidance on how to achieve maximum performance.
- Appendix A lists all CUDA-enabled devices.
- Appendix B is a detailed description of all extensions to the C language.
- Appendix C lists the mathematical functions supported in CUDA.
- Appendix D lists the C++ features supported in device code.
- Appendix E gives more details on texture fetching.
- Appendix F gives the technical specifications of various devices, as well as more architectural details.

Chapter 2. Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. An extensive description of CUDA C is given in Chapter 3.

Full code for the vector addition example used in this chapter and the next can be found in the *vectorAdd* SDK code sample.

2.1 Kernels

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

A kernel is defined using the **`__global__`** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **`<<<`**...**`>>>`** *execution configuration* syntax (see Appendix B.16). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **`threadIdx`** variable.

As an illustration, the following sample code adds two vectors *A* and *B* of size *N* and stores the result into vector *C*:

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

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

Here, each of the *N* threads that execute **`VecAdd()`** performs one pair-wise addition.

2.2 Thread Hierarchy

For convenience, **`threadIdx`** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional *thread block*. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + y D_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + y D_x + z D_x D_y)$.

As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

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

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks as illustrated by Figure 2-1. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

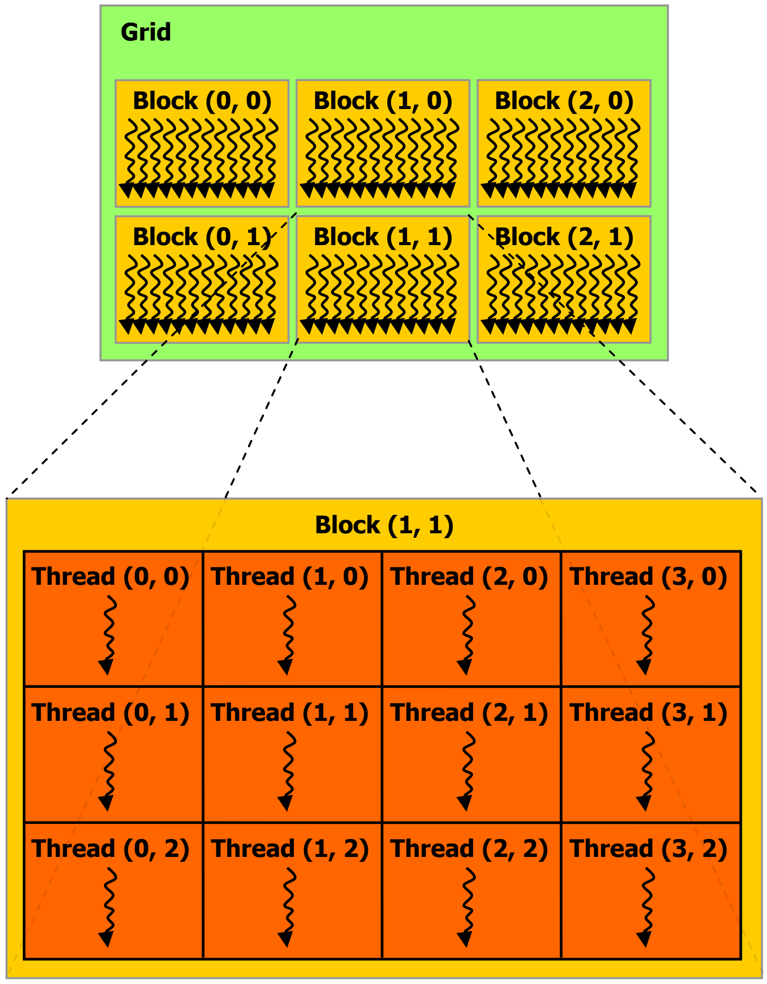


Figure 2-1. Grid of Thread Blocks

The number of threads per block and the number of blocks per grid specified in the **<<<…>>>** syntax can be of type **`int`** or **`dim3`**. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in **`blockIdx`** variable. The dimension of the thread block is accessible within the kernel through the built-in **`blockDim`** variable.

Extending the previous **`MatAdd()`** example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 1-4, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the **`__syncthreads()`** intrinsic function; **`__syncthreads()`** acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Section 3.2.3 gives an example of using shared memory.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and **`__syncthreads()`** is expected to be lightweight.

2.3 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2-2. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. 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. The global, constant, and texture memory spaces are optimized for different memory usages (see Sections 5.3.2.1, 5.3.2.4, and 5.3.2.5). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 3.2.10).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

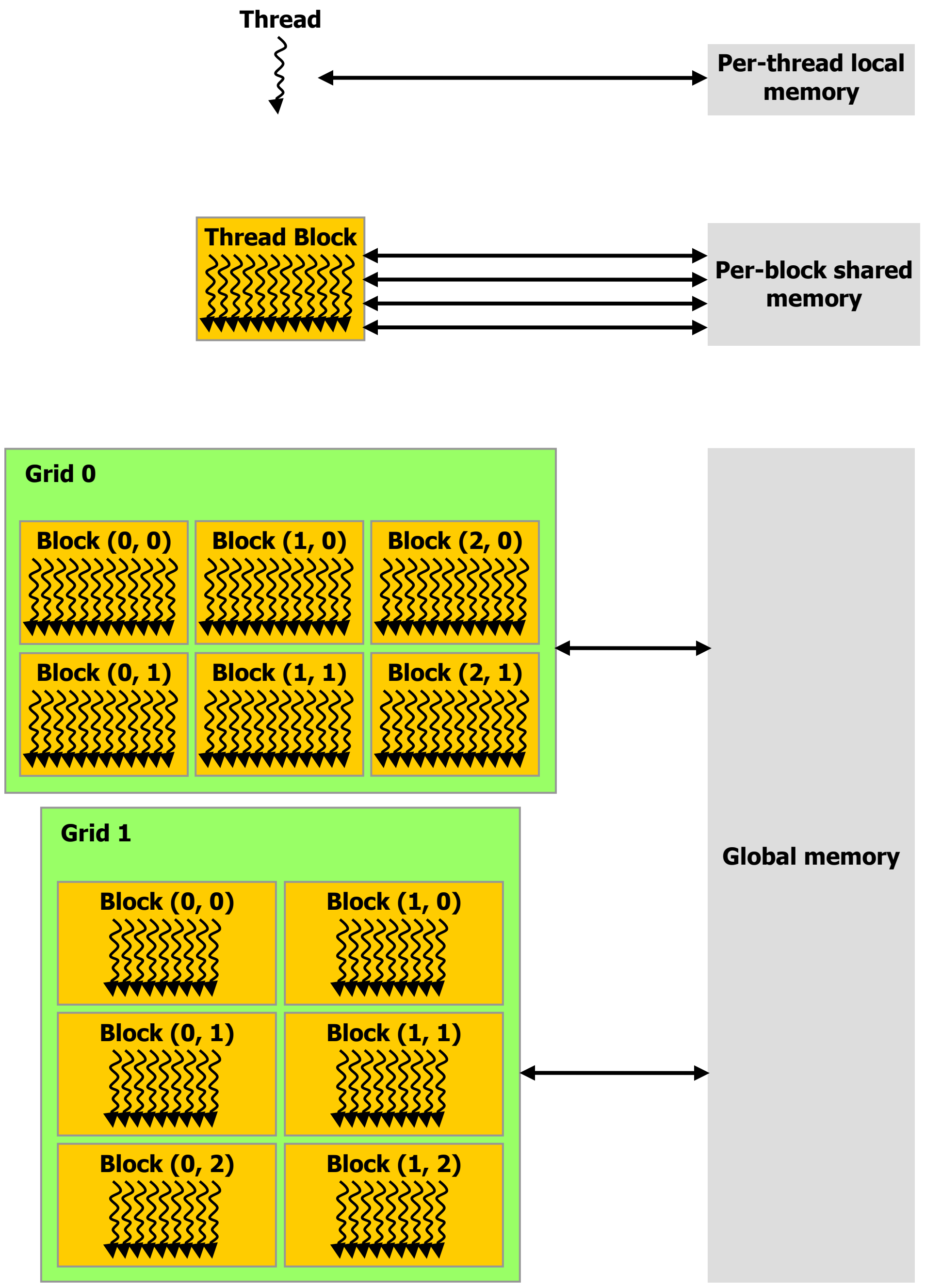
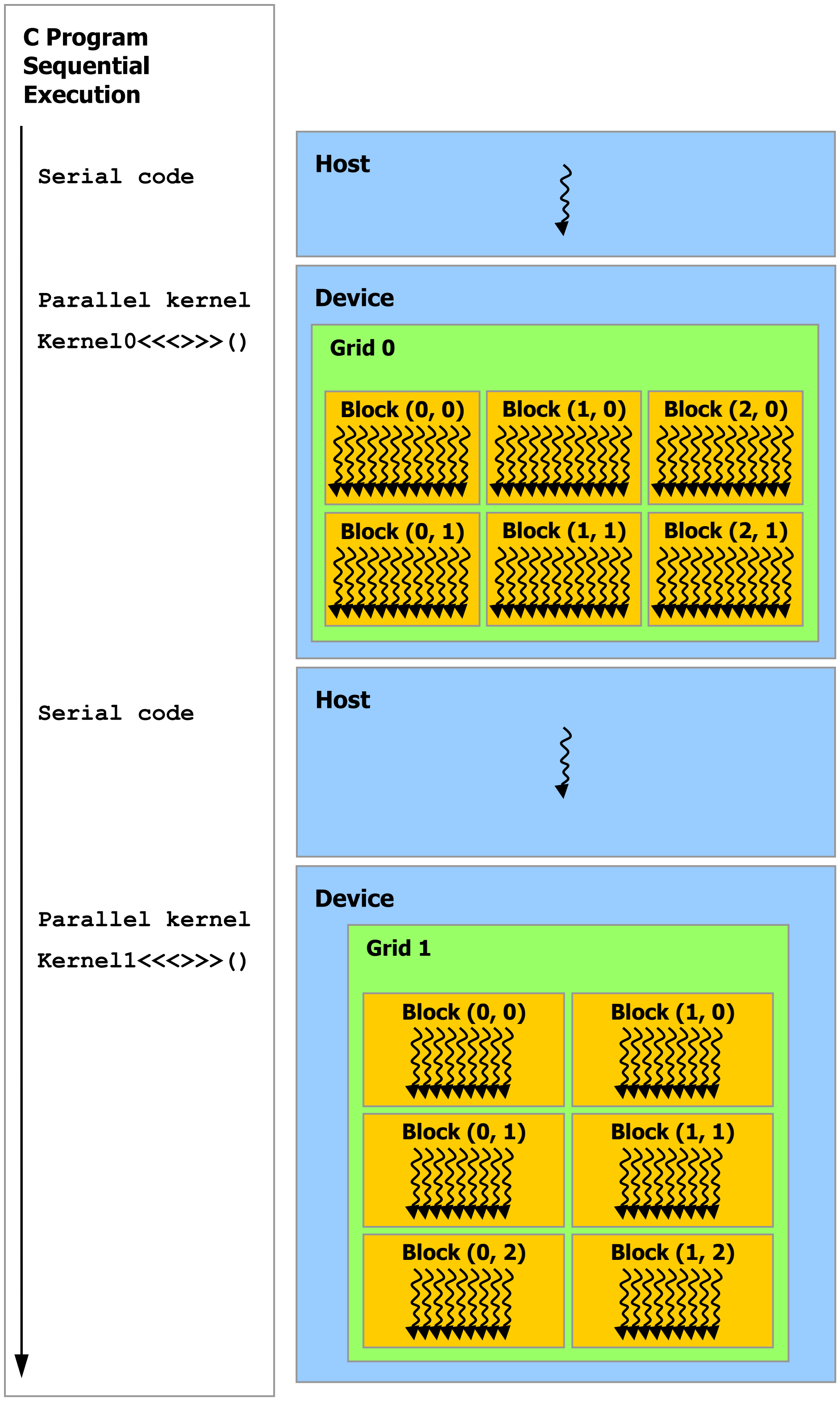


Figure 2-2. Memory Hierarchy

2.4 Heterogeneous Programming

As illustrated by Figure 2-3, the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in Chapter 3). This includes device memory allocation and deallocation as well as data transfer between host and device memory.



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

2.5 Compute Capability

The *compute capability* of a device is defined by a major revision number and a minor revision number.

Devices with the same major revision number are of the same core architecture. The major revision number of devices based on the Fermi architecture is 2. Prior devices are all of compute capability 1.x (Their major revision number is 1).

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

Appendix A lists of all CUDA-enabled devices along with their compute capability. Appendix F gives the technical specifications of each compute capability.

Appendix B.
C Language Extensions

B.1 Function Type Qualifiers

Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

B.1.1 __device__

The **`__device__`** qualifier declares a function that is:

- Executed on the device
- Callable from the device only.

B.1.2 __global__

The **`__global__`** qualifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host only.

`__global__` functions must have **`void`** return type.

Any call to a **`__global__`** function must specify its execution configuration as described in Section B.16.

A call to a **`__global__`** function is asynchronous, meaning it returns before the device has completed its execution.

B.1.3 __host__

The **`__host__`** qualifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the **`__host__`** qualifier or to declare it without any of the **`__host__`**, **`__device__`**, or **`__global__`** qualifier; in either case the function is compiled for the host only.

The **`__global__`** and **`__host__`** qualifiers cannot be used together.

The **`__device__`** and **`__host__`** qualifiers can be used together however, in which case the function is compiled for both the host and the device. The **`__CUDA_ARCH__`** macro introduced in Section 3.1.4 can be used to differentiate code paths between host and device:

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ == 100
    // Device code path for compute capability 1.0
#elif __CUDA_ARCH__ == 200
    // Device code path for compute capability 2.0
#elif !defined(__CUDA_ARCH__)
    // Host code path
#endif
}
```

B.1.4 __noinline__ and __forceinline__

When compiling code for devices of compute capability 1.x, a **`__device__`** function is always inlined by default. When compiling code for devices of compute capability 2.x, a **`__device__`** function is only inlined when deemed appropriate by the compiler.

The **`__noinline__`** function qualifier can be used as a hint for the compiler not to inline the function if possible. The function body must still be in the same file where it is called. For devices of compute capability 1.x, the compiler will not honor the **`__noinline__`** qualifier for functions with pointer parameters and for functions with large parameter lists. For devices of compute capability 2.x, the compiler will always honor the **`__noinline__`** qualifier.

The **`__forceinline__`** function qualifier can be used to force the compiler to inline the function.

B.2 Variable Type Qualifiers

Variable type qualifiers specify the memory location on the device of a variable.

An automatic variable declared in device code without any of the **`__device__`**, **`__shared__`** and **`__constant__`** qualifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in Section 5.3.2.2.

B.2.1 __device__

The **`__device__`** qualifier declares a variable that resides on the device.

At most one of the other type qualifiers defined in the next three sections may be used together with **`__device__`** to further specify which memory space the variable belongs to. If none of them is present, the variable:

- Resides in global memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library (**`cudaGetSymbolAddress()`** / **`cudaGetSymbolSize()`** / **`cudaMemcpyToSymbol()`** / **`cudaMemcpyFromSymbol()`** for the runtime API and **`cuModuleGetGlobal()`** for the driver API).

B.2.2 __constant__

The **`__constant__`** qualifier, optionally used together with **`__device__`**, declares a variable that:

- Resides in constant memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library (**`cudaGetSymbolAddress()`** / **`cudaGetSymbolSize()`** / **`cudaMemcpyToSymbol()`** / **`cudaMemcpyFromSymbol()`** for the runtime API and **`cuModuleGetGlobal()`** for the driver API).

B.2.3 __shared__

The **`__shared__`** qualifier, optionally used together with **`__device__`**, declares a variable that:

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Is only accessible from all the threads within the block.

When declaring a variable in shared memory as an external array such as

```
extern __shared__ float shared[];
```

the size of the array is determined at launch time (see Section B.16). All variables declared in this fashion, start at the same address in memory, so that the layout of the variables in the array must be explicitly managed through offsets. For example, if one wants the equivalent of

```
short array0[128];
float array1[64];
int   array2[256];
```

in dynamically allocated shared memory, one could declare and initialize the arrays the following way:

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int*   array2 =   (int*)&array1[64];
}
```

Note that pointers need to be aligned to the type they point to, so the following code, for example, does not work since **array1** is not aligned to 4 bytes.

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[127];
}
```

Alignment requirements for the built-in vector types are listed in Table B-1.

B.2.4 __restrict__

nvcc supports restricted pointers via the **__restrict__** keyword.

Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

Here is an example subject to the aliasing issue, where use of restricted pointer can help the compiler to reduce the number of instructions:

```
void foo(const float* a,
         const float* b,
         float* c)
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

In C-type languages, the pointers **a**, **b**, and **c** may be aliased, so any write through **c** could modify elements of **a** or **b**. This means that to guarantee functional correctness, the compiler cannot load **a[0]** and **b[0]** into registers, multiply them, and store the result to both **c[0]** and **c[1]**, because the results would differ from the abstract execution model if, say, **a[0]** is really the same location as **c[0]**. So the compiler cannot take advantage of the common sub-expression. Likewise, the compiler cannot just reorder the computation of **c[4]** into the proximity of the computation of **c[0]** and **c[1]** because the preceding write to **c[3]** could change the inputs to the computation of **c[4]**.

By making **a**, **b**, and **c** restricted pointers, the programmer asserts to the compiler that the pointers are in fact not aliased, which in this case means writes through **c** would never overwrite elements of **a** or **b**. This changes the function prototype as follows:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c);
```

Note that all pointer arguments need to be made restricted for the compiler optimizer to derive any benefit. With the **__restrict** keywords added, the compiler can now reorder and do common sub-expression elimination at will, while retaining functionality identical with the abstract execution model:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c)
{
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t2;
    float t3 = a[1];
    c[0] = t2;
    c[1] = t2;
    c[4] = t2;
    c[2] = t2 * t3;
    c[3] = t0 * t3;
    c[5] = t1;
    ...
}
```

The effects here are a reduced number of memory accesses and reduced number of computations. This is balanced by an increase in register pressure due to "cached" loads and common sub-expressions.

Since register pressure is a critical issue in many CUDA codes, use of restricted pointers can have negative performance impact on CUDA code, due to reduced occupancy.

B.3 Built-in Vector Types

B.3.1 char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, longlong1, ulonglong1, longlong2, ulonglong2, float1, float2, float3, float4, double1, double2

These are vector types derived from the basic integer and floating-point types. They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields `x`, `y`, `z`, and `w`, respectively. They all come with a constructor function of the form **`make_<type name>`**; for example,

```
int2 make_int2(int x, int y);
```

which creates a vector of type **`int2`** with value **`(x, y)`**.

In host code, the alignment requirement of a vector type is equal to the alignment requirement of its base type. This is not always the case in device code as detailed in Table B-1.

Table B-1. Alignment Requirements in Device Code

Type	Alignment
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4, ulong4	16
longlong1, ulonglong1	8
longlong2, ulonglong2	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

B.3.2 dim3

This type is an integer vector type based on **`uint3`** that is used to specify dimensions. When defining a variable of type **`dim3`**, any component left unspecified is initialized to 1.

B.4 Built-in Variables

Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device.

B.4.1 gridDim

This variable is of type **`dim3`** (see Section B.3.2) and contains the dimensions of the grid.

B.4.2 blockIdx

This variable is of type **`uint3`** (see Section B.3.1) and contains the block index within the grid.

B.4.3 blockDim

This variable is of type **`dim3`** (see Section B.3.2) and contains the dimensions of the block.

B.4.4 threadIdx

This variable is of type **`uint3`** (see Section B.3.1) and contains the thread index within the block.

B.4.5 warpSize

This variable is of type **`int`** and contains the warp size in threads (see Section 4.1 for the definition of a warp).

B.5 Memory Fence Functions

```
void __threadfence_block();
```

waits until all global and shared memory accesses made by the calling thread prior to **`__threadfence_block()`** are visible to all threads in the thread block.

```
void __threadfence();
```

waits until all global and shared memory accesses made by the calling thread prior to **`__threadfence()`** are visible to:

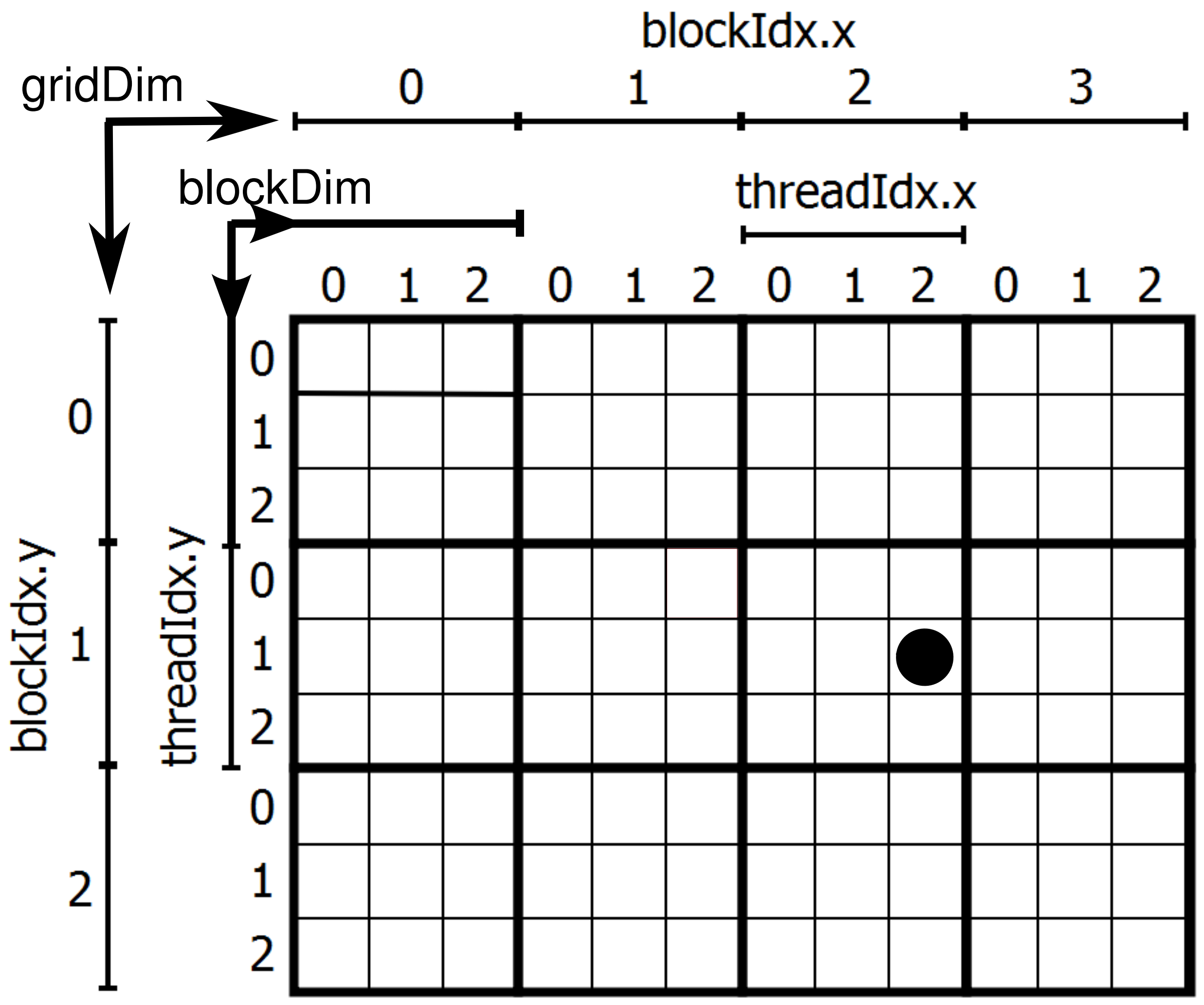
- All threads in the thread block for shared memory accesses,
- All threads in the device for global memory accesses.

```
void __threadfence_system();
```

waits until all global and shared memory accesses made by the calling thread prior to **`__threadfence_system()`** are visible to:

- All threads in the thread block for shared memory accesses,
- All threads in the device for global memory accesses,
- Host threads for page-locked host memory accesses (see Section 3.2.4.3).

Identifikace vlákna - globální proměnné



x = blockIdx.x * blockDim.x + threadIdx.x
y = blockIdx.y * blockDim.y + threadIdx.y

`__threadfence_system()` is only supported by devices of compute capability 2.x.

In general, when a thread issues a series of writes to memory in a particular order, other threads may see the effects of these memory writes in a different order. **`__threadfence_block()`**, **`__threadfence()`**, and **`__threadfence_system()`** can be used to enforce some ordering.

One use case is when threads consume some data produced by other threads as illustrated by the following code sample of a kernel that computes the sum of an array of N numbers in one call. Each block first sums a subset of the array and stores the result in global memory. When all blocks are done, the last block done reads each of these partial sums from global memory and sums them to obtain the final result. In order to determine which block is finished last, each block atomically increments a counter to signal that it is done with computing and storing its partial sum (see Section B.11 about atomic functions). The last block is the one that receives the counter value equal to **`gridDim.x-1`**. If no fence is placed between storing the partial sum and incrementing the counter, the counter might increment before the partial sum is stored and therefore, might reach **`gridDim.x-1`** and let the last block start reading partial sums before they have been actually updated in memory.

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    float* result)
{
    // Each block sums a subset of the input array
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {

        // Thread 0 of each block stores the partial sum
        // to global memory
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure its result is visible to
        // all other threads
        __threadfence();

        // Thread 0 of each block signals that it is done
        unsigned int value = atomicInc(&count, gridDim.x);

        // Thread 0 of each block determines if its block is
        // the last block to be done
        isLastBlockDone = (value == (gridDim.x - 1));
    }

    // Synchronize to make sure that each thread reads
    // the correct value of isLastBlockDone
    __syncthreads();

    if (isLastBlockDone) {

        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);

        if (threadIdx.x == 0) {

            // Thread 0 of last block stores total sum
            // to global memory and resets count so that
            // next kernel call works properly
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

B.6 Synchronization Functions

```
void __syncthreads();
```

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to **`__syncthreads()`** are visible to all threads in the block.

`__syncthreads()` is used to coordinate communication between the threads of the same block. When some threads within a block access the same addresses in shared or global memory, there are potential read-after-write, write-after-read, or write-after-write hazards for some of these memory accesses. These data hazards can be avoided by synchronizing threads in-between these accesses.

`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

Devices of compute capability 2.x support three variations of **`__syncthreads()`** described below.

```
int __syncthreads_count(int predicate);
```

is identical to **`__syncthreads()`** with the additional feature that it evaluates **`predicate`** for all threads of the block and returns the number of threads for which **`predicate`** evaluates to non-zero.

```
int __syncthreads_and(int predicate);
```

is identical to **`__syncthreads()`** with the additional feature that it evaluates **`predicate`** for all threads of the block and returns non-zero if and only if **`predicate`** evaluates to non-zero for all of them.

```
int __syncthreads_or(int predicate);
```

is identical to **`__syncthreads()`** with the additional feature that it evaluates **`predicate`** for all threads of the block and returns non-zero if and only if **`predicate`** evaluates to non-zero for any of them.

B.7 Mathematical Functions

Appendix C gives the list of all C/C++ standard library mathematical functions that are supported in device code and all intrinsic functions that are only supported in device code, along with their respective error bounds.

more details). All counters are reset before each kernel call (note that when an application is run via a CUDA debugger or profiler (cuda-gdb, CUDA Visual Profiler, Parallel Nsight), all launches are synchronous).

B.14 Formatted Output

Formatted output is only supported by devices of compute capability 2.x.

```
int printf(const char *format[, arg, ...]);
```

prints formatted output from a kernel to a host-side output stream.

The in-kernel **printf()** function behaves in a similar way to the standard C-library **printf()** function, and the user is referred to the host system's manual pages for a complete description of **printf()** behavior. In essence, the string passed in as **format** is output to a stream on the host, with substitutions made from the argument list wherever a format specifier is encountered. Supported format specifiers are listed below.

The **printf()** command is executed as any other device-side function: per-thread, and in the context of the calling thread. From a multi-threaded kernel, this means that a straightforward call to **printf()** will be executed by every thread, using that thread's data as specified. Multiple versions of the output string will then appear at the host stream, once for each thread which encountered the **printf()**.

It is up to the programmer to limit the output to a single thread if only a single output string is desired (see Section B.14.4 for an illustrative example).

Unlike the C-standard **printf()**, which returns the number of characters printed, CUDA's **printf()** returns the number of arguments parsed. If no arguments follow the format string, 0 is returned. If the format string is NULL, -1 is returned. If an internal error occurs, -2 is returned.

B.14.1 Format Specifiers

As for standard **printf()**, format specifiers take the form:

%[flags][width][.precision][size]type

The following fields are supported (see widely-available documentation for a complete description of all behaviors):

- *Flags*: `'#'` `' '` `'0'` `'+'` `'-'`
- *Width*: `'*'` `'0-9'`
- *Precision*: `'0-9'`
- *Size*: `'h'` `'l'` `'ll'`
- *Type*: `'%cdiouxXpeEfgGaAs'`

Note that CUDA's **printf()** will accept *any* combination of flag, width, precision, size and type, whether or not overall they form a valid format specifier. In other words, "*%hd*" will be accepted and printf will expect a double-precision variable in the corresponding location in the argument list.

B.14.2 Limitations

Final formatting of the **`printf()`** output takes place on the host system. This means that the format string must be understood by the host-system's compiler and C library. Every effort has been made to ensure that the format specifiers supported by CUDA's printf function form a universal subset from the most common host compilers, but exact behavior will be host-O/S-dependent.

As described in Section B.14.1, **`printf()`** will accept *all* combinations of valid flags and types. This is because it cannot determine what will and will not be valid on the host system where the final output is formatted. The effect of this is that output may be undefined if the program emits a format string which contains invalid combinations.

The **`printf()`** command can accept at most 32 arguments in addition to the format string. Additional arguments beyond this will be ignored, and the format specifier output as-is.

Owing to the differing size of the **`long`** type on 64-bit Windows platforms (four bytes on 64-bit Windows platforms, eight bytes on other 64-bit platforms), a kernel which is compiled on a non-Windows 64-bit machine but then run on a win64 machine will see corrupted output for all format strings which include **`"%ld"`**. It is recommended that the compilation platform matches the execution platform to ensure safety.

The output buffer for **`printf()`** is set to a fixed size before kernel launch (see Section B.14.3). It is circular and if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. It is flushed only when one of these actions is performed:

- Kernel launch via **`<<<>>>`** or **`cuLaunchKernel()`** (at the start of the launch, and if the CUDA_LAUNCH_BLOCKING environment variable is set to 1, at the end of the launch as well),
- Synchronization via **`cudaDeviceSynchronize()`**, **`cuCtxSynchronize()`**, **`cudaStreamSynchronize()`**, **`cuStreamSynchronize()`**, **`cudaEventSynchronize()`**, or **`cuEventSynchronize()`**,
- Memory copies via any blocking version of **`cudaMemcpy*()`** or **`cuMemcpy*()`**,
- Module loading/unloading via **`cuModuleLoad()`** or **`cuModuleUnload()`**,
- Context destruction via **`cudaDeviceReset()`** or **`cuCtxDestroy()`**.

Note that the buffer is not flushed automatically when the program exits. The user must call **`cudaDeviceReset()`** or **`cuCtxDestroy()`** explicitly, as shown in the examples below.

B.14.3 Associated Host-Side API

The following API functions get and set the size of the buffer used to transfer the **`printf()`** arguments and internal metadata to the host (default is 1 megabyte):

- Driver API:

cuCtxGetLimit(size_t* size, CU_LIMIT_PRINTF_FIFO_SIZE)
cuCtxSetLimit(CU_LIMIT_PRINTF_FIFO_SIZE, size_t size)

- Runtime API:
  cudaDeviceGetLimit(size_t* size,cudaLimitPrintfFifoSize)
  cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)

B.14.4 Examples

The following code sample:

```
// printf() is only supported
// for devices of compute capability 2.0 and above
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
    #define printf(f, ...) ((void)(f, __VA_ARGS__),0)
#endif

__global__ void helloCUDA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

void main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceReset();
}
```

will output:

```
Hello thread 2, f=1.2345
Hello thread 1, f=1.2345
Hello thread 4, f=1.2345
Hello thread 0, f=1.2345
Hello thread 3, f=1.2345
```

Notice how each thread encounters the **printf()** command, so there are as many lines of output as there were threads launched in the grid. As expected, global values (i.e. **float f**) are common between all threads, and local values (i.e. **threadIdx.x**) are distinct per-thread.

The following code sample:

```
__global__ void helloCUDA(float f)
{
    if (threadIdx.x == 0)
        printf("Hello thread %d, f=%f\n", threadIdx.x, f) ;
}

void main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceReset();
}
```

will output:

```
Hello thread 0, f=1.2345
```