Getting started with CUDA

Edwin Carlinet, Joseph Chazalon {firstname.lastname@lrde.epita.fr}

November'19

EPITA Research & Development Laboratory (LRDE)

Sources and license

Much of this lesson is based on these great resources — *Look them up to go further!*

- The course "GPU Teaching Kit", 2019, licensed by NVidia and the University of Illinois under the Creative Commons Attribution-Non Commercial 4.0 International License.
- The book "Programming massively parallel processors" (Third Edition), D. Kirk and W. Hwu, Elsevier, 2017.
- The manual "CUDA C Programming Guide", NVidia, v10.1.243 (August 19, 2019).

Lesson overview

CUDA overview

Host view of GPU computation

Compilation and Runtime

Kernel programming

Debugging, Performance analysis and Profiling

CUDA overview

What is CUDA?

A product

- It enables to use NVidia GPUs for computation

A C variant

- Mostly C++14-compatible, with extensions
- and also some restrictions!

A SDK

- A set of compilers and toolchains for various architectures
- Performance analysis tools

A runtime

- An assembly specification
- Computation libraries (linear algebra, etc.)

A new industry standard

- Used by every major deep learning framework
- Replacing OpenCL as Vulkan is replacing OpenGL

The CUDA ecosystem

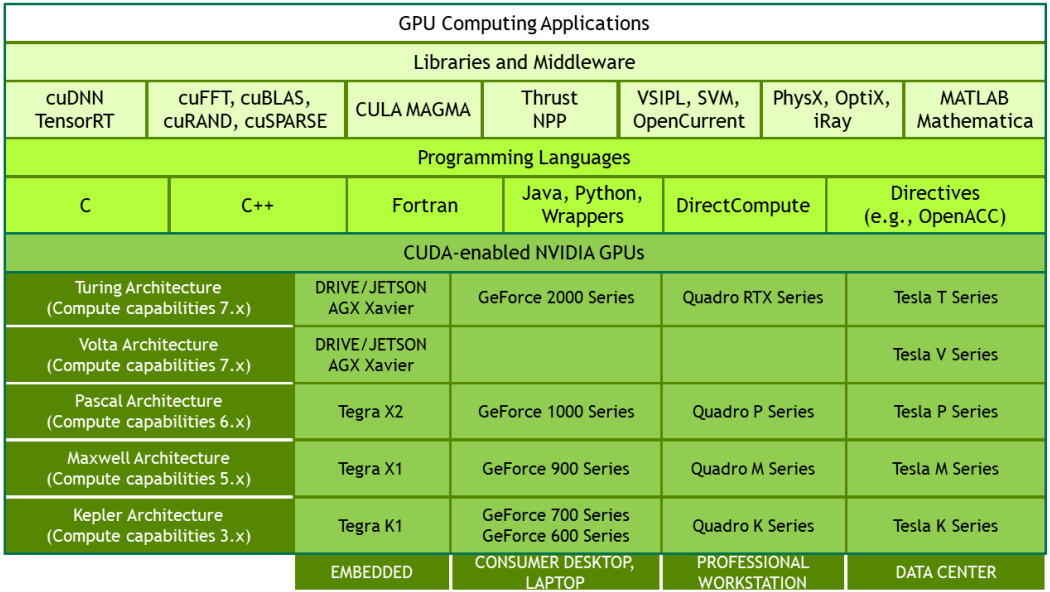

Figure 1: The CUDA ecosystem

Libraries *or* Compiler Directives *or* Programming Language?

CUDA is mostly based on a "new" **programming language**: CUDA C (or C++, or Fortran).
This grants much flexibility and performance

But is also exposes much of GPU goodness through **libraries**.

And it supports a few **compiler directives** to facilitate some constructs.

```
#pragma unroll
for(int i = 0; i < WORK_PER_THREAD; ++i)
  // Some thread work
```

The big idea: Kernels instead of loops

Without CUDA (vector addition)

```
// compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  for (int i = 0; i < n; ++i)
    h_C[i] = h_A[i] + h_B[i];
}

int main()
{
  // Allocation for A, B and C
  // I/O to read n elements of A and B
  vecAdd(h_A, h_B, h_C);
}
```

With CUDA (1/2): move work to the separate compute device

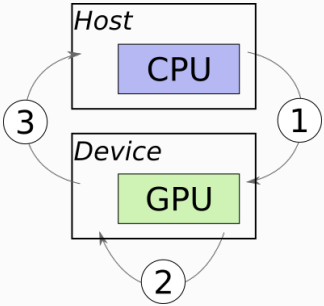

Figure 2: Computation on separate device

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  // 1.1 Allocate device memory for A, B and C
  // 1.2 Copy A and B to device memory

  // 2. Launch kernel code - computation done on device

  // 3. Copy C (result) from device memory
  // Free device vectors
}

int main() { /* Unchanged */ }
```

With CUDA (2/2): Kernel sample code

```
// kernel
__global__ void kvecAdd(float *d_A, float *d_B, float *d_C, int n)
{
  int i = blockDim.x * blockID.x + threadId.x;
  if (i >= n) return;
  d_C[i] = d_A[i] + d_B[i];
}
```

No more `for` loop!

Arrays of parallel threads

A CUDA kernel is executed by a **grid** (array) of threads

- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that is uses to compute memory addresses and make control decisions

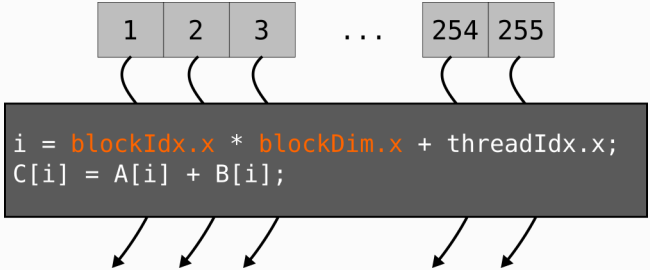

Figure 3: A thread block

Thread blocks

Threads are grouped into thread blocks

- Threads within a block cooperate via
  - **shared memory**
  - **atomic operations**
  - **barrier synchronization**
- Threads in different blocks do not interact[1]

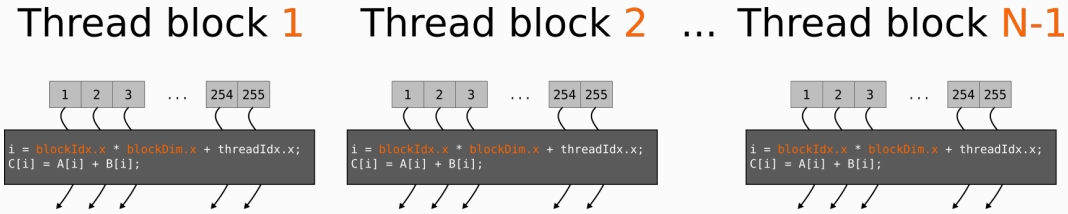

Figure 4: Independent thread blocks

[1] Not in this course, though there are techniques for that.

A multidimensional grid of computation threads (1/2)

Each thread uses indices to decide what data to work on:

- `blockIdx` ($0 \rightarrow$ `gridDim`): 1D, 2D or 3D
- `threadIdx` ($0 \rightarrow$ `blockDim`): 1D, 2D or 3D

Each index has `x`, `y` and `z` attributes to get the actual index in each dimension.

```
int i = threadIdx.x;
int j = threadIdx.y;
int k = threadIdx.z;
```

Simplifies memory addressing when processing multidimensional data:

- *image processing*
- *solving PDE on volumes*
- ...

A multidimensional grid of computation threads (2/2)

Grid and blocks can have different dimensions,
but they usually are two levels of the same work decomposition.

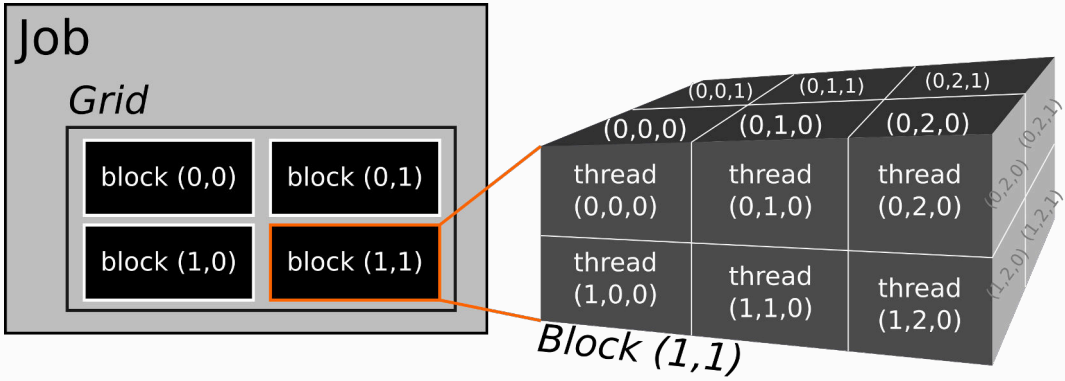

Figure 5: An example of 2D grid with 3D blocks

Grid & block examples (1/2)

Vector addition (N elements)

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

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);  // <-- So this is how we launch CUDA kernels!
    ...
}
```

Grid & block examples (2/2)

Matrix addition (N×N elements)

```
// 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]; /* Missing boundary check. */
}

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);
    ...
}
```

Block decomposition enable automatic scalability

Because the work is divided into **independent blocs** which can be **run in parallel** on each *streaming multiprocessor* (SM),
the same code can be **automatically scaled** to architectures with more or less SMs...

as long as SMs architectures are compatibles (100% compatible with the same Compute Capabilities version — a family of devices, careful otherwise).

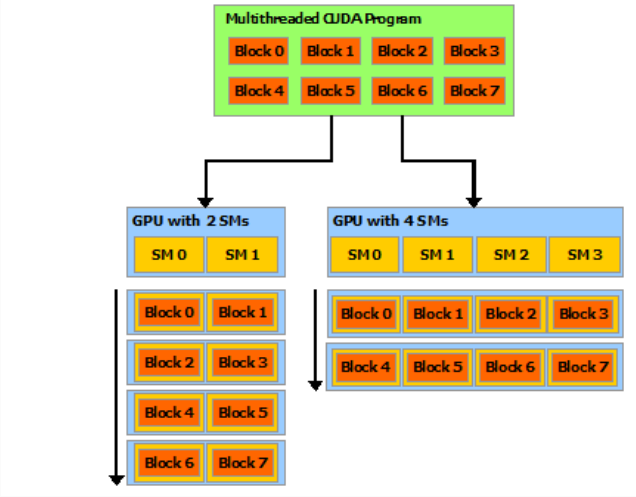

Figure 6: Automatic scaling

Building and running a simple program

CUDA Hello world (`hello.cu`)

```
#include <stdio.h>

__global__ void print_kernel() {
    printf(
      "Hello from block %d, thread %d\n",
      blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<2, 3>>>();
    cudaDeviceSynchronize();
}
```

Compile

```
$ nvcc hello.cu -o hello
```

Run

```
$ ./hello
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 0, thread 0
Hello from block 0, thread 1
Hello from block 0, thread 2
```

What you need to get started

NVidia GPU hardware

NVidia GPU drivers, properly loaded
- *`modprobe nvidia ...`*

CUDA runtime libraries
- *`libcuda.so`, `libnvidia-fatbinaryloader.so`, ...*

CUDA SDK (NVCC compiler in particular)
- *relies on a standard C/C++ compiler and toolchain*
- *docs.nvidia.com/cuda/cuda-installation-guide-linux*

Basic C/C++ knowledge

Summary

Host vs Device $\leftrightarrow$ Separate memory

GPUs are computation units which require explicit usage, as opposed to a CPU

Need to load data to and fetch result from device

Replace loops with kernels

Kernel = Function computed in relative isolation on small chunks of data, on the GPU

Divide the work

Problem $\rightarrow$ Grid $\rightarrow$ Blocs $\rightarrow$ Threads

Compile and run using CUDA SDK

`nvcc`, `libcuda.so`, ...

Host view of GPU computation

Calling kernels, not writing them

You do not need to write kernels to run CUDA code:
you can use kernels from a library, written by someone else.

This section is about how to properly launch CUDA kernels using their API only.

Sequential and parallel sections

We use the GPU(s) as co-processor(s).

Our program is made of a series of sequential and parallel sections.

Of course, CPU code can be multi-threaded too!

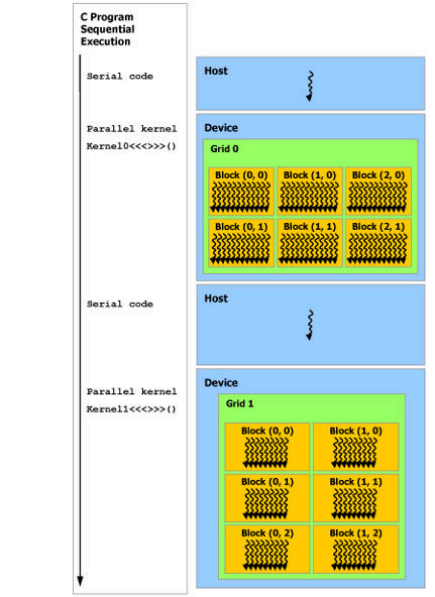

Figure 7: Heterogeneous programming

Host *vs* device: reminder

We need to transfer inputs from the host to the device and outputs the other way around.

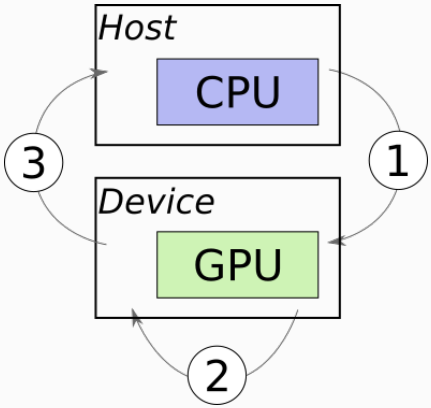

Figure 8: Computation on separate device

A proper kernel invocation

Let's fix this code!

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  // 1.1 Allocate device memory for A, B and C
  // 1.2 Copy A and B to device memory

  // 2. Launch kernel code - computation done on device

  // 3. Copy C (result) from device memory
  // Free device vectors
}
```

CUDA memory primitives

	1D	2D	3D
Allocate	`cudaMalloc()`	`cudaMallocPitch()`	`cudaMalloc3D()`
Copy	`cudaMemcpy()`	`cudaMemcpy2D()`	`cudaMemcpy3D()`
On-device init.	`cudaMemset()`	`cudaMemset2D()`	`cudaMemset3D()`
Reclaim		`cudaFree()`	

. . . plus many others detailed in the CUDA Runtime API documentation. . .

Why 2D and 3D variants?

- Strong alignment requirements in device memory
  - Enables correct loading of memory chunks to SM caches (correct bank alignment)
- Proper striding management in automated fashion

Host ↔ Device memory transfer

We just need the three following ones for now:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size_in_bytes )
```

Allocates space in the **device global** memory.

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t size_in_bytes,
                         cudaMemcpyKind kind )
```

Asynchronous data transfer. `cudaMemcpyKind` ≈ copy direction:

- `cudaMemcpyHostToHost` ← **useful**
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost` ← **useful**
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyDefault` ← *Direction inferred from pointer values. Requires unified virtual addressing.*

```
cudaError_t cudaFree ( void* devPtr )
```

Almost complete code

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;
  // 1.1 Allocate device memory for A, B and C
  cudaMalloc((void **) &d_A, size); // TODO repeat for d_B and d_C
  // 1.2 Copy A and B to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  // 2. Launch kernel code - computation done on device
  k_VecAdd<<<NB, NT>>>(d_A, d_B, d_C);  // FIXME How to compute NB and NT?
  // 3. Copy C (result) from device memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // Free device vectors
  cudaFree(d_A); // TODO repeat for d_B and d_C
}
```

Checking errors

In practice, we need to check for API errors

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess)  {
    printf("%s in %s at line %d\n",
           cudaGetErrorString(err),
           __FILE__,
           __LINE__);
    exit(EXIT_FAILURE);
}
```

Intermission: *Can I use memory management functions inside kernels?*

No: `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()` shall be called from host only.

However, kernels may allocate, use and reclaim memory dynamically using regular `malloc()`, `memset()`, `memcpy()` and `free()` functions.

Note that if some device code allocates some memory, it must free it.

Fix the kernel invocation line

We want to fix this line:

```
k_VecAdd<<<NB, NT>>>(d_A, d_B, d_C);
```

Kernel invocation syntax:

```
kernel<<<blocks, threads_per_block, shmem, stream>>>(param1, param2, ...);
```

- `blocks`: number of blocks in the grid;
- `threads_per_block`: number of threads for each block;
- `shmem`: (opt.) amount of shared memory to allocate (in bytes);
- `stream`: (opt.) CUDA stream (no discussed in this lesson, see the documentation).

How to set gridDim and BlockDim properly?

Lvl 0: Naive trial with as many threads as possible

```
k_VecAdd<<<1, size>>>(d_A, d_B, d_C);
```

How to set gridDim and BlockDim properly?

Lvl 0: Naive trial with as many threads as possible

```
k_VecAdd<<<1, size>>>(d_A, d_B, d_C);
```

Will fail with large vectors.

Hardware limitation on the maximum number of threads per block (1024 for Compute Capability 3.0-7.5).

Will fail with vectors of size which is not a multiple of warp size.

Lvl 1: It works with just enough blocks

```
// Get max threads per block
int devId = 0; // There may be more devices!
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, devId);
printf("Maximum grid dimensions: %d x %d x %d\n",
        deviceProp.maxGridSize[0],
        deviceProp.maxGridSize[1],
        deviceProp.maxGridSize[2]);
printf("Maximum block dimensions: %d x %d x %d\n",
        deviceProp.maxThreadsDim[0],
        deviceProp.maxThreadsDim[1],
        deviceProp.maxThreadsDim[2]);
// Compute the number of blocks
int xThreads = deviceProp.maxThreadsDim[0];
dim3 DimBlock(xThreads, 1, 1); // 1D VecAdd
int xBlocks =  (int) ceil(n/xThreads);
dim3 DimGrid(xBlocks, 1, 1);
// Launch the kernel
k_VecAdd<<<DimGrid, DimBlock>>>(d_A, d_B, d_C);
```

Lvl 2: Tune block size given kernel requirements and hardware constraints
It is important to understand the difference between:

- the logical decomposition of your program:
  problem $\approx$ grid $\rightarrow$ blocks $\rightarrow$ threads
- the scheduling of the computation on the hardware:
  - assignment of each block to a *Streaming Multiprocessor* (SM)
  - groups threads into *warps*
  - run groups of *warps* concurrently

The hardware constraints are different between each *Compute Capability* version. See the CUDA C programming manual, Appendix H for details about each hardware version.
In particular, the amount of memory available on each SM may limit the number of threads one would actually want to launch...

But this depends on the kernel code!
The **CUDA Occupancy Calculator** APIs are designed to assist programmers in choosing the best number of threads per block based on register and shared memory requirements of a given kernel.

But wait...

```
#include <stdio.h>
__global__ void print_kernel() {
    printf("Hello!\n");
}

int main() {
    print_kernel<<<1, 1>>>();

}
```

This code prints nothing!

Kernel invocation is asynchronous

```
#include <stdio.h>
__global__ void print_kernel() {
    printf("Hello!\n");
}

int main() {
    print_kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

Host code synchronization requires `cudaDeviceSynchronize()` because **kernel invocation is asynchronous** from host perspective.

On the device, kernel invocations are strictly sequential (unless you schedule them on different *streams*).

Intermission: *Can I call kernels inside kernels?*

Yes: This is the basis of *dynamic parallelism*.

Some restrictions over the stack size apply.
Remember that the device runtime is a functional subset of the host runtime, ie you can perform device management, kernel launching, device memcpy, etc., but with some restrictions (see the documentation for details).

The compiler may inline some of those calls, though.

Conclusion about the host-only view

A host-only view of the computation is sufficient for most of the cases:

1. upload input data to the device
2. fire a kernel
3. download output data from the device

Advanced CUDA requires to make sure we saturate the SMs, and may imply some kernel study to determine the best:

- amount of threads per blocks
- amount of blocks per grid
- work per thread (if applicable)
- ...

This depends on:

- hardware specifications: maximum `gridDim` and `blockDim`, etc.
- kernel code: amount of register and shared memory used by each thread

Compilation and Runtime

Compilation simplified overview

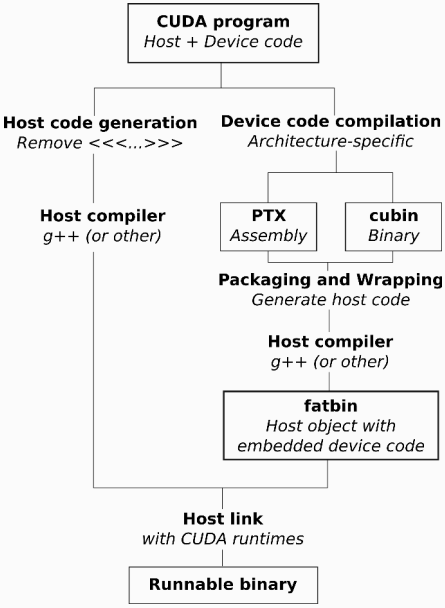

Figure 9: Separate compilation of host and device code

Host and devices code follow two different compilation trajectories.

Device code is compiled into two formats:

- **PTX assembly** tied to a *virtual architecture* specification
- **cubin binary code** tied to a particular GPU product family — aka *real architecture* like *Fermi*, *Kepler*, *Maxwell*, *Pascal*, *Volta*, *Turing* and *Ampere* (soon)

The final runnable binary

- contains both host and device code
- is linked with the CUDA runtime(s).

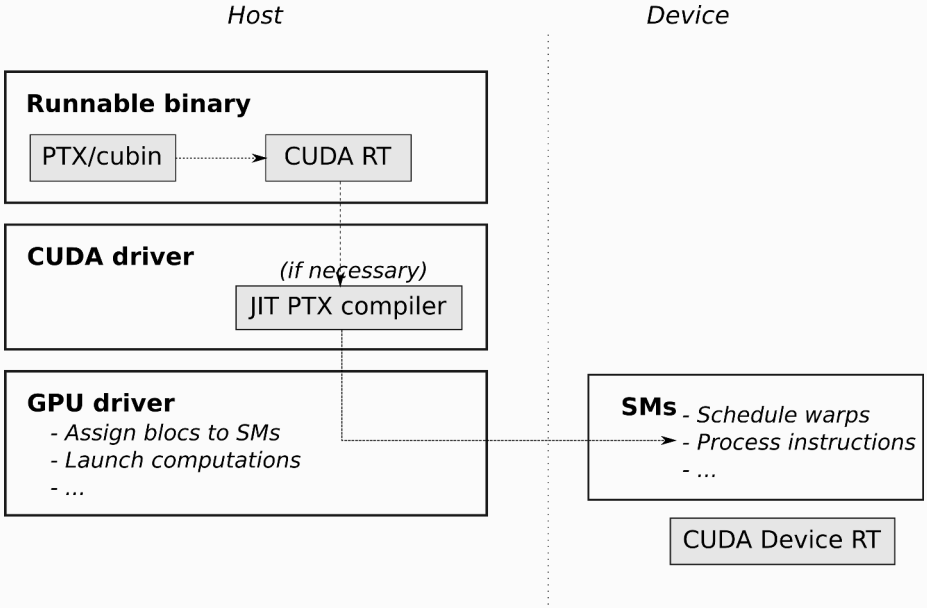

Figure 10: Transfer of code to device with optional JIT compilation

PTX, cubin, fatbinary... Why?

Because NVidia wants to be able to push innovations on their hardware as soon as possible, they **do not ensure forward compatibility of binaries**, unlike CPU vendors.

They break forward compatibility at each major GPU release, ie when they release a new GPU family.

Real architectures vs Virtual architectures

Real architectures

Hardware version	Features
sm_30 and sm_32	Basic features + Kepler support + Unified memory programming
sm_35	+ Dynamic parallelism support
sm_50, sm_52 and sm_53	+ Maxwell support
sm_60, sm_61 and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support

Virtual architectures

Compute capability	Features
compute_30 and compute_32	Basic features + Kepler support + Unified memory programming
compute_35	+ Dynamic parallelism support
compute_50, compute_52, and compute_53	+ Maxwell support
compute_60, compute_61, and compute_62	+ Pascal support
compute_70 and compute_72	+ Volta support
compute_75	+ Turing support

Real architectures ("code")

- Run compiled binary code (cubin)
- Instantiate a virtual architecture to a particular number of SMs per GPU
- Specifies a particular SM model
- Noted `sm_xx`
- Selected using the `-code` parameter of `nvcc`

What's the point?

- Pre-compile your kernels for a particular hardware and accelerate program startup.

Virtual architectures ("arch")

- Specifies an instruction set for PTX assembly (ptx) (much like SSE extensions)
- Specifies features available
- Noted `compute_xx`
- Selected using the `-arch` parameter of `nvcc`

What's the point?

- Limit the features you want to use to maximize compatibility
- Migrate code progressively as some behavior may change (like Independent Thread Scheduling in `compute_70`)
- The `__CUDA_ARCH__` macro will be set accordingly in your code so you can have different code paths for different compute capabilities

More on compute capabilities

Excellent summaries:

- Appendix H on Compute Capabilities of CUDA C programming guide
- CUDA page on Wikipedia
- List of GPUs and their compute capability version available here: developer.nvidia.com/cuda-gpus

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	**3.0**	**3.2**	**3.5, 3.7, 5.0, 5.2**	**5.3**	**6.x**	**7.x**
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())	Yes					
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())	Yes					
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes					
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes					
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes					
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No				Yes	
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming						
Funnel shift (see reference manual)	No	Yes				
Dynamic Parallelism	No		Yes			
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No			Yes		
Tensor Core	No					Yes

Figure 11: Feature Support per Compute Capability

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32			16		128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	2^{31}-1											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [45]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB			
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 ~ 128 KB	32 or 64 KB
Maximum width for a 1D texture reference bound to a CUDA array	65536											
Maximum width for a 1D texture reference bound to linear memory	2^{27}											
Maximum width and number of layers for a 1D layered texture reference	16384 x 2048											
Maximum width and height for a 2D texture reference bound to a CUDA array	65536 x 65535											
Maximum width and height for a 2D texture reference bound to linear memory	65000 x 65000											
Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather	16384 x 16384											
Maximum width, height, and number of layers for a 2D layered texture reference	16384 x 16384 x 2048											
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	4096 x 4096 x 4096											
Maximum width (and height) for a cubemap texture reference	16384											
Maximum width (and height) and number of layers for a cubemap layered texture reference	16384 x 2046											
Maximum number of textures that can be bound to a kernel	256											
Maximum width for a 1D surface reference bound to a CUDA array	65536											
Maximum width and number of layers for a 1D layered surface reference	65536 x 2048											
Maximum width and height for a 2D surface reference bound to a CUDA array	65536 x 32768											
Maximum width, height, and number of layers for a 2D layered surface reference	65536 x 32768 x 2048											
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array	65536 x 32768 x 2048											
Maximum width (and height) for a cubemap surface reference bound to a CUDA array	32768											
Maximum width (and height) and number of layers for a cubemap layered surface reference	32768 x 2046											
Maximum number of surfaces that can be bound to a kernel	16											
Maximum number of instructions per kernel	512 million											

Figure 12: Technical Specifications per Compute Capability

Documentation excerpt

Compute Capability 3.x:

- **Architecture**

  A multiprocessor consists of:
  - 192 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 32 special function units for single-precision floating-point transcendental functions,
  - 4 warp schedulers.
- **Global Memory**
  - Global memory accesses for devices of compute capability 3.x are cached in L2...
  - A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory...
- **Shared Memory**
  - Shared memory has 32 banks...

Compute Capability 5.x:

- **Architecture**

  A multiprocessor consists of:
  - 128 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 32 special function units for single-precision floating-point transcendental functions,
  - 4 warp schedulers.
- ...

Binary code compatibility (cubin)

GPU (device) binary code is **not forward (nor backward) compatible**:
it is architecture-specific and can be run only by hardware with the same major version.

Example:
Binary code compiled and optimized for sm_30 cards

- can be run by sm_32 and sm_35 cards (Kepler family),
- but cannot be run by sm_5x cards (Maxwell family).

PTX code compatibility

Assembly code, however, is based on an always-increasing set of instructions (much like SSE extensions).
This implies two things:

- **PTX assembly is forward compatible with newer architectures**,
- it is **not backward compatible** though,
- it is always possible to compile
  the PTX assembly of an earlier version (like `compute_30`)
  to a binary for the most recent architecture (like `sm_75`).

This is how NVidia ensures that old code will still run on newer hardware.
New code, however, will not run on old hardware unless special care is taken (more on that later).

CUDA Driver and PTX compilation

The **CUDA driver** (`libcuda.so`) contains the **JIT PTX compiler** and is **always backward compatible** (this is what actually makes PTX forward compatible).

This means that it can take assembly code from an older version anc compile it for the current version of the device on the current machine.

However, it is **not forward compatible**: code compiled with newer PTX assembly cannot be understood.

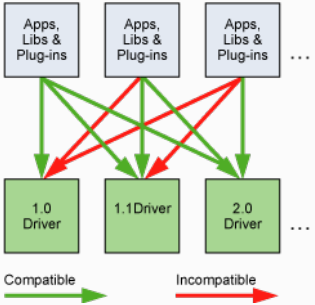

Figure 13: Compatibility of CUDA Versions

It may be necessary to ask the user to install a newer version of the CUDA driver on its system.

CUDA Runtime and SDK support

The **CUDA runtime** (`libcudart.so`) is bundled with your SDK an provides high-level functionnality.

- You should distribute the CUDA runtime with your application.
- It is compatible with a certain range of GPU driver versions.
- It supports a certain range of hardware (GPU families):
  - ...
  - CUDA SDK 6.5 support for compute capability 1.1 - 5.x (Tesla, Fermi, Kepler, Maxwell). Last version with support for compute capability 1.x (Tesla)
  - CUDA SDK 7.0 - 7.5 support for compute capability 2.0 - 5.x (Fermi, Kepler, Maxwell)
  - CUDA SDK 8.0 support for compute capability 2.0 - 6.x (Fermi, Kepler, Maxwell, Pascal). Last version with support for compute capability 2.x (Fermi)
  - CUDA SDK 9.0 - 9.2 support for compute capability 3.0 - 7.2 (Kepler, Maxwell, Pascal, Volta)
  - CUDA SDK 10.0 - 10.2 support for compute capability 3.0 - 7.5 (Kepler, Maxwell, Pascal, Volta, Turing). Last version with support for compute capability 3.x (Kepler)

Two-stage compilation

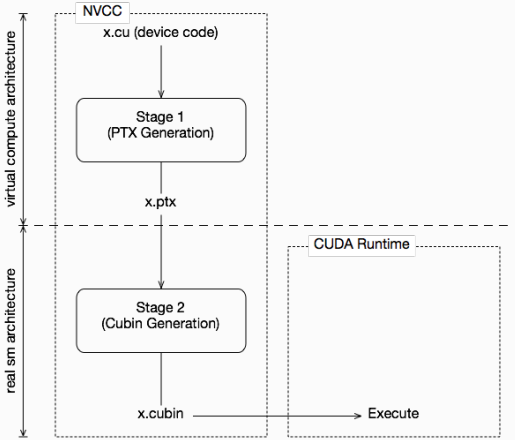

Figure 14: Two-Staged (offline) Compilation with Virtual and Real Architectures

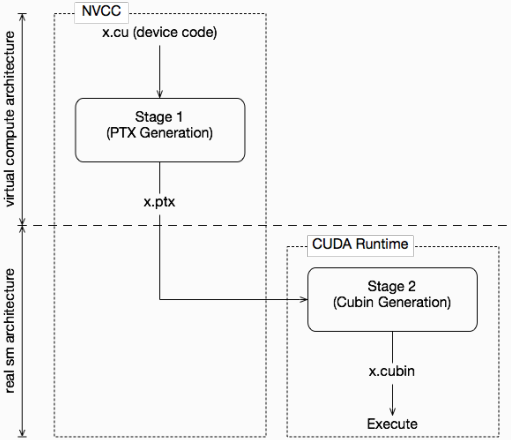

Figure 15: Just-in-Time Compilation of Device Code

The complete compilation trajectory

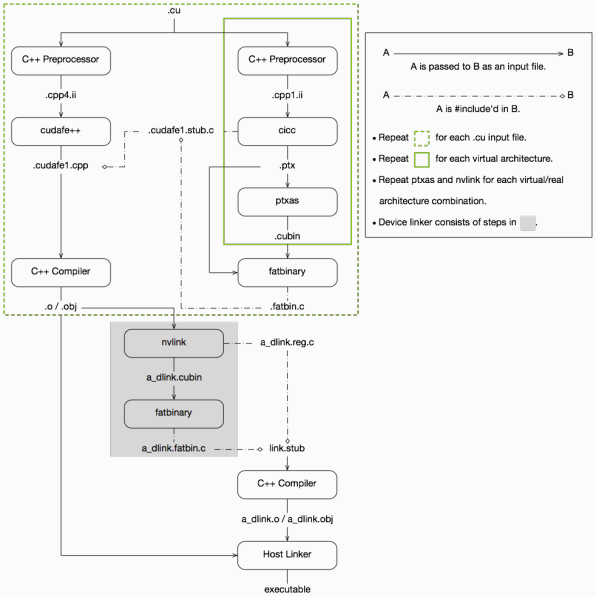

Figure 16: CUDA compilation trajectory

Whole program compilation vs Separate compilation (of device code)

Separate compilation of source code is possible.

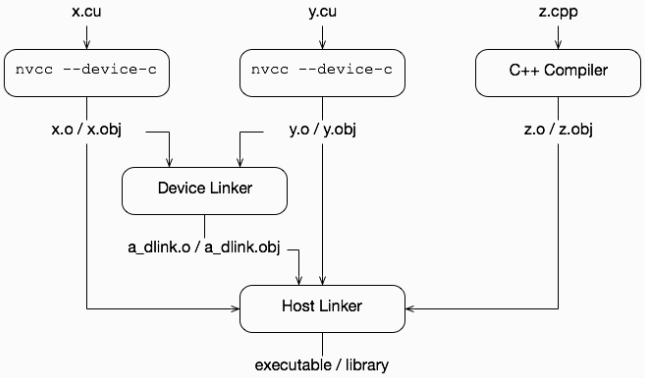

Figure 17: CUDA Separate Compilation Trajectory

As of Nov. 2019, what is safe to use?

Maximum compatibility

```
/usr/local/cuda/bin/nvcc
  -gencode=arch=compute_30,code=sm_30
  -gencode=arch=compute_35,code=sm_35
  -gencode=arch=compute_50,code=sm_50
  -gencode=arch=compute_60,code=sm_60
  -gencode=arch=compute_70,code=sm_70
  -gencode=arch=compute_75,code=sm_75
  -gencode=arch=compute_75,code=compute_75
  -O2 -o mykernel.o -c mykernel.cu
```

Distribute the cudart lib (static or dynamic link) with your application.

`__CUDA_ARCH__`

Use different code paths to support previous architectures.

```
__device__ func()
{
#if __CUDA_ARCH__ < 350
  /* Do something special for
  architectures without dynamic
  parallelism. */
#else
  /* Do something else. */
#endif
}
```

Deprecations

Kepler and Maxwell hardware will be deprecated (sm_3x, sm_5x) in the next CUDA versions.

Compilation and Runtime Summary

Host code and device code are compiled separately.

- Device code is packaged with host code to be launched.
- A host compiler (ex $g++$) is required.

You can select which features you want to activate in your code,
hence which compatibility you offer.

- Using `__CUDA_ARCH__` macro in your code to support multiple architectures.
- Using `nvcc`'s `-arch compute_xx` flag.
- This controls the PTX assembly which is generated.
- PTX assembly is forward compatible thanks to JIT compilation.

You can select the hardware you want to build a precompiled binary (cubin) for.

- Accelerates application startup (do not care about it for now).
- Using `nvcc`'s `-code sm_xx` flag.

You can generate multiples PTX and cubins using the following `nvcc`'s flags repeatidly:
`-gencode arch=compute_xx,code=sm_yy`

Kernel programming

(Reminder) 3 simple abstractions for a scalable programming model

CUDA is based at its core on 3 key abstractions:

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

This enables a CUDA program to be:

- partitionned in blocks
- run on devices with different computation resources

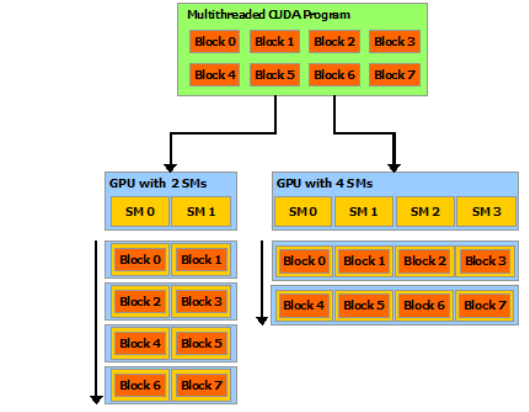

Figure 18: Automatic scaling

Several API levels

We now want to program kernels.
There are several APIs available:

- PTX assembly
- Driver API (C)
- Runtime C++ API ← **let us use this one**

We will first focus on the **language extensions** added to support kernel programming.
They are described in detail in Appendix B of the CUDA C Programming Guide.

Function Execution Space Specifiers

		Executed on the:	Only callable from the:
`__host__`	`float HostFunc()`	host	host
`__global__`	`void KernelFunc()`	device	host*
`__device__`	`float DeviceFunc()`	device	device

- `__global__` defines a kernel function
  - Each "`__`" consists of two underscore characters
  - A kernel function must return void
  - *It may be called from another kernel for devices of compute capability 3.2 or higher (Dynamic Parallelism support)
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Built-in Vector Types (1/2)

They make is easy to work with data like images.
Alignement must be respected in all operations.

Type	Align.
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8

Type	Align.
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

Type	Align.
longlong1, ulonglong1	8
longlong2, ulonglong2	16
longlong3, ulonglong3	8
longlong4, ulonglong4	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16
double3	8
double4	16

Built-in Vector Types (2/2)

They all are structures.

They all come with a constructor function of the form `make_<type name>`:

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

The 1st, 2nd, 3rd, and 4th components are accessible through the fields x, y, z, and w, respectively.

```
uint4 p = make_uint4(128, 128, 128, 255);
// or uint4 p(128, 128, 128, 255);
uint r = p.x, g = p.y, b = p.z, a = p.w;
```

`dim3` is an alias of `uint3` for which any component left unspecified is initialized to 1. Used to specify grid and block sizes.

```
dim3 blockSize(32, 32);
```

Built-in Variables

Some variables are pre-defined in a kernel and can be used directly.

Name	Type	Description
gridDim	dim3	dimensions of the grid
blockIdx	uint3	block index within the grid
blockDim	dim3	dimensions of the block
threadIdx	uint3	thread index within the block
warpSize	int	warp size in threads

Example:

```
__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]; /* Missing boundary check. */
}
```

Memory Hierarchy

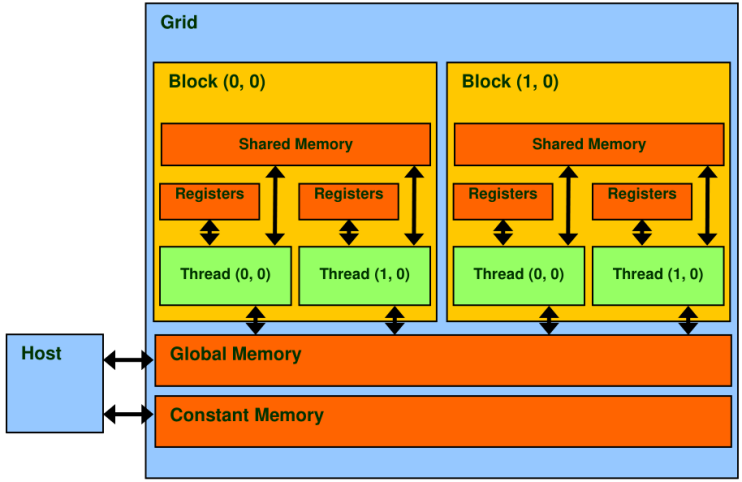

Figure 19: Programmer view of CUDA memories

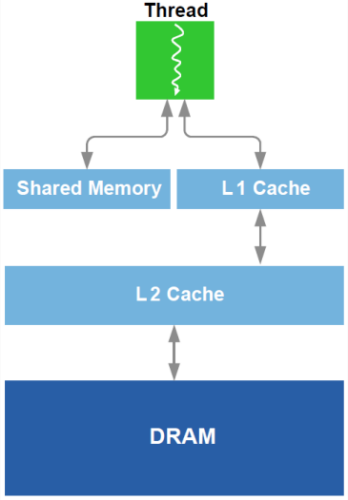

Figure 20: Cache hierarchy

Types of Memory

Registers Used to store parameters, local variables, etc.
Very fast
Private to each thread
Lots of threads $\implies$ little memory per thread (spills in global memory if needed)

Shared Used to store temporary data
Very fast
Shared among all threads in a block

Constant A special cache for read-only values
Slow at first then very fast

Global Large and slow
Shared among all threads in all blocks (in all kernels)

Caches Transparent use

Local *Local thread memory cached to L2 and/or L1*
Ultimately stored in global memory if needed

Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes[‡]	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	Yes[†]	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation

[†] Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

[‡] Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Cost to Access Memory

Space	Time	Notes
Register	0	
Shared	0	
Constant	0	Amortized cost is low, first access is high
Local	$>$ 100 clocks	
Parameter	0	
Global	$>$ 100 clocks	

Variable Memory Space Specifiers

How to declaring CUDA variables

Variable declaration		Memory	Scope	Lifetime
`int LocalVar;`		register	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

Remarks:

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register

Where to declare variables?

Can host access it?

- Yes: **global** and **constant**
  Declare outside of any function
- No: **register** and **shared**
  Use or declare in the kernel

Example: Shared Variable Declaration

```
__global__ MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // ...
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  // ...
}
```

Can also be declared to use dynamically allocated memory.
See the documentation for further details.

What can be shared by who?

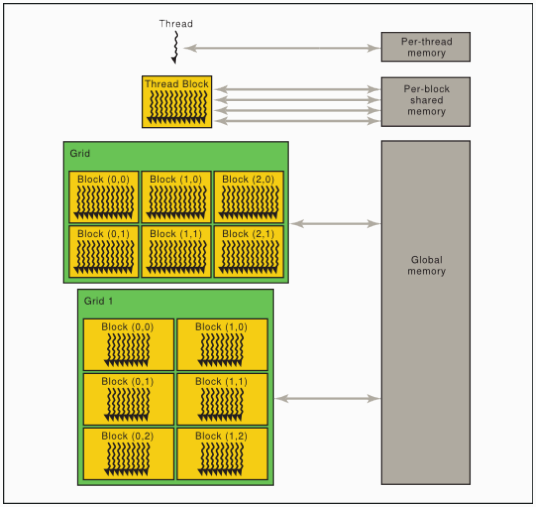

Figure 21: Memory sharing among threads, blocks and grids

Possible memory access:

- Among threads in the same grid (a kernel invocation):
  - Global memory
- Among threads in the same block:
  - Global memory
  - Shared memory (efficient)
- Per threads:
  - Global (not efficient)
  - Shared memory
  - Registers and local

Relaxed consistency memory model

The CUDA programming model assumes a device with a **weakly-ordered memory model**, that is the order in which a CUDA thread writes data to shared memory or global memory, is not necessarily the order in which the data is observed being written by another CUDA or host thread.

Example:

```
__device__ volatile int X = 1, Y = 2;
__device__ void write_from_thread1()
{
    X = 10;
    Y = 20;
}
```

```
__device__ void read_from_thread2()
{
    int A = X;
    int B = Y;
}
```

Possible outcomes for thread 2

Strongly-ordered memory model:

- A = 1 and B = 2
- A = 10 and B = 2
- A = 10 and B = 20

Weakly-ordered memory model (like CUDA):

- All the previous
- And also A = 1 and B = 20!

Memory Fence Functions

Memory fence functions can be used to enforce some ordering on memory accesses.

```
void __threadfence_block();
```

ensures that:

- All writes to all memory made by the calling thread before the call to `__threadfence_block()` are observed by all threads in the block of the calling thread as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_block()`;
- All reads from all memory made by the calling thread before the call to `__threadfence_block()` are ordered before all reads from all memory made by the calling thread after the call to `__threadfence_block()`.

Like a flush of read and write queues.

```
void __threadfence();
```

acts as `__threadfence_block()` but also ensure that threads from others blocks observe writes in order. **This requires to read an uncached value** and implies the use of the `volatile` keywords.

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.

Stronger than `__threadfence()` because it also synchronizes the execution.

`__syncthreads()` is used to coordinate communication between the threads of the same block.

`__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.

Atomic Functions (1/2)

Atomic functions perform a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

Most of the atomic functions are available for all the numerical types:
`int`, `unsigned int`, `unsigned long long int`, `float`, `double`, `half`, etc.

Arithmetic functions

```
int atomicAdd(int* address, int val);
//int atomicSub(int* address, int val);
```

Read `old` at `address`, computes `(old + val)` and stores it back to `address`, returns `old`.

```
int atomicExch(int* address, int val);
```

Read `old` at `address`, stores `val` to `address`, and returns `old`.

```
int atomicMin(int* address, int val);
// int atomicMax(int* address, int val);
```

Compute and store min (max).

Atomic Functions (2/2)

Arithmetic functions (cont'd)

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
//unsigned int atomicDec(unsigned int* address, unsigned int val);
```

Computes `(((old == 0) || (old > val)) ? val : (old-1)`

```
int atomicCAS(int* address, int compare, int val);
```

Computes `(old == compare ? val : old)`

Bitwise functions

```
int atomicAnd(int* address, int val);
int atomicOr(int* address, int val);
int atomicXor(int* address, int val);
```

Long example: Tiled matrix multiplication

TODO next lesson

Debugging, Performance analysis and Profiling

printf

Possible since Fermi devices (Compute Capability 2.x and higher).

Limited amount of lines:

- circular buffer flushed at particular times)
- but **not** at program exit: must include call to `cudaDeviceSynchronize()` before exiting

Example:

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

int main() {
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

OUTPUT:

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

Global memory write

To dump then inspect a larger amount of intermediate data.
Analysis code should be removed for production.

Example:

```
__global__ void mykernel(float *input, float *output, float *intermediate) {
    // ...
    intermediate[threadIdx.x] = intermediate_result;
    // ...
}

int main() {
    // allocate input, output AND intermediate
    // ...
    mykernel<<<GS, BS>>>(input, output, intermediate);
    // ...
    // analyse intermediate results
    // ...
}
```

Check error messages

Did you check the error codes?

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess)  {
    printf("%s in %s at line %d\n",
           cudaGetErrorString(err),
           __FILE__,
           __LINE__);
    exit(EXIT_FAILURE);
}
```

CUDA tools

CUDA-GDB debugger

Debugging flags:

- `-g`: include host debugging information
- `-G`: include device debugging information
- `-lineinfo`: include line information with symbols

Based on GDB.

CUDA-MEMCHECK memory debugging tool

- No recompilation necessary
  `cuda-memcheck myprogram`
- Can detect the following errors: memory leaks, memory errors (like alignment issues), race conditions, illegal barriers...

nvprof profiler

`nvprof myprogram`

NSight

- Visual tool
- Great visualization of profiling results
- Other tools integrated

Other tools

- *cuobjdump*: host and device obj disassemble and overview
- *nvdisasm*: advanced analysis of device binaries
- *nvprune*: prunes host object files and libraries to only contain device code for the specified targets