

# Top-Down CUDA Overview

Changho Hwang  
chhwang@kaist.ac.kr

# CUDA Installation for Ubuntu – 1

- Install dependencies
  - `sudo apt-get install gcc make`
- Visit <https://developer.nvidia.com/cuda-downloads> and download a proper runfile (local)
  - The runfile name is like `cuda_x.x.xx_linux.run`
- Install NVIDIA driver and CUDA toolkit
  - Run `chmod u+x cuda_x.x.xx_linux.run`
  - Run `sudo ./cuda_x.x.xx_linux.run`
  - **Accept** the EULA
  - Install graphics driver: **y**
  - Install OpenGL libraries: **n** if you are not sure
  - Run `nvidia-xconfig`: **n** for the best GPGPU performance
  - Install CUDA toolkit: **y**
  - Enter toolkit location: type enter (follow the default)
  - Install a symbolic link: **y**
  - Install CUDA samples: **y** if you need sample codes
- If the installation succeeds, go to the next slide
- If the installation fails, check logs in `/var/log/nvidia-installer.log`
  - `ERROR: Unable to load the kernel module 'nvidia.ko'`  
→ Make sure that secure boot mode is disabled from BIOS. Disable it and try again.

**Select Target Platform** ⓘ

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OSX
Architecture ⓘ	x86_64	ppc64le	
Distribution	Fedora	OpenSUSE	RHEL CentOS SLES Ubuntu
Version	16.04	14.04	
Installer Type ⓘ	runfile (local)	deb (local)	deb (network) cluster (local)

**Download Installer for Linux Ubuntu 16.04 x86\_64**

The base installer is available for download below.

**> Base Installer** Download [1.4 GB] ⬇

Installation Instructions:

1. Run `sudo sh cuda_8.0.44_linux.run`
2. Follow the command-line prompts

The CUDA Toolkit contains Open-Source Software. The source code can be found [here](#).  
 The checksums for the installer and patches can be found in [Installer Checksums](#).  
 For further information, see the [Installation Guide for Linux](#) and the [CUDA Quick Start Guide](#).

# CUDA Installation for Ubuntu – 2

- If you are installing for all users (recommended)
  - Run `sudo vi /etc/environment`
  - Add `/usr/local/cuda/bin` in `PATH`
  - Add `LD_LIBRARY_PATH=/usr/local/cuda/lib64`
  - Save the file and run `source /etc/environment`
  - Example of `/etc/environment`

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/local/cuda/bin"
LD_LIBRARY_PATH=/usr/local/cuda/lib64
```

- If you are installing for your account only (not recommended)
  - Run `vi ~/.bashrc` and at the end of the file,
  - Add `export PATH="$PATH:/usr/local/cuda/bin"`
  - Add `export LD_LIBRARY_PATH=/usr/local/cuda/lib64`
  - Save the file and run `source ~/.bashrc`

```
~$ nvidia-smi
Sun Jan 15 15:17:04 2017
```

NVIDIA-SMI 367.48 Driver Version: 367.48												
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC	Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.
0	GeForce GTX TIT...	Off	0000:05:00.0	Off	N/A	22%	52C	P0	72W / 250W	0MiB / 12203MiB	0%	Default
1	GeForce GTX TIT...	Off	0000:09:00.0	Off	N/A	0%	47C	P0	53W / 250W	0MiB / 12206MiB	0%	Default

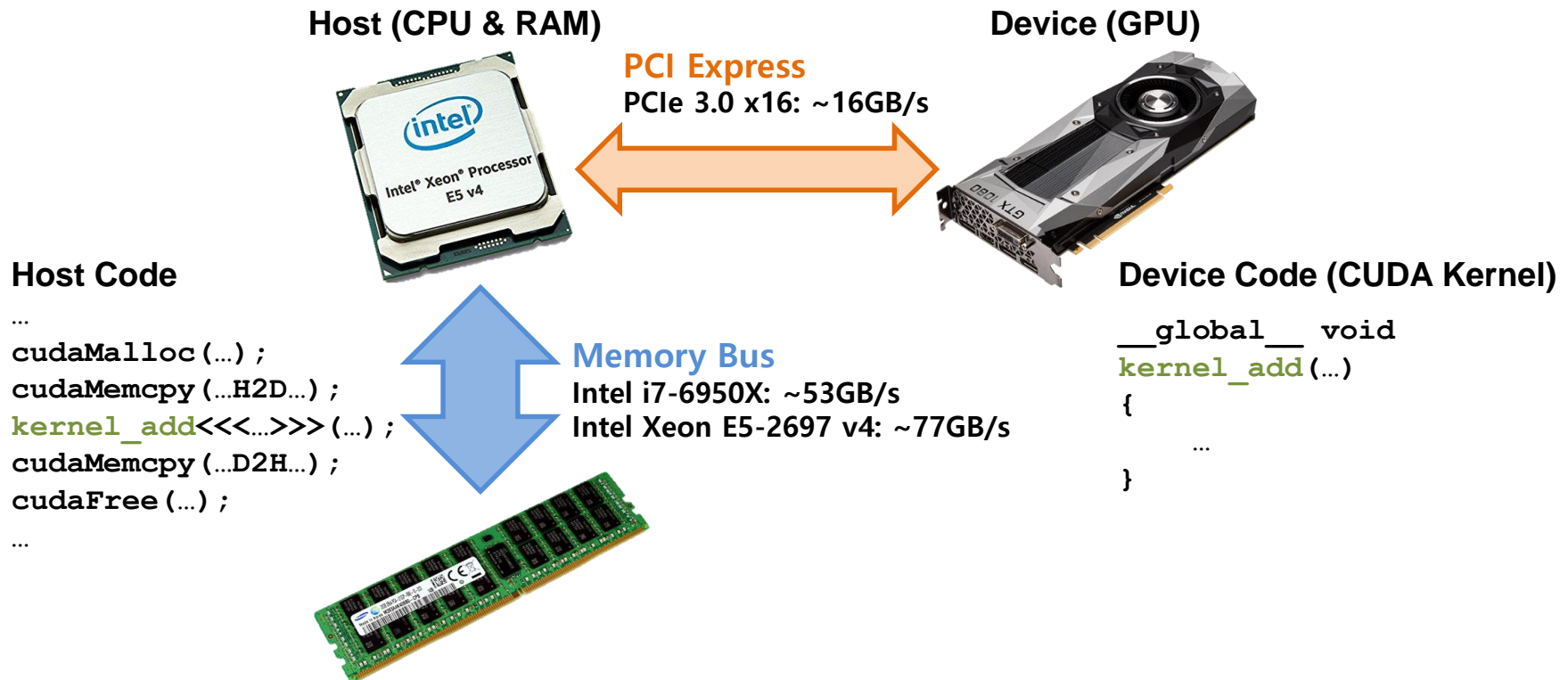
  

Processes:					GPU Memory
GPU	PID	Type	Process name		Usage
No running processes found					

- Done ! (some CUDA applications may require reboot)
- If some problems occur, check whether the driver module is loaded
  - Run `nvidia-smi` and check whether it shows GPU info as the picture
  - `modprobe: FATAL: Module nvidia not found`  
 → You may have enabled secure boot after the installation.  
 Disable secure boot from BIOS and re-install the *graphics driver only again* via  
`sudo ./cuda_x.x.xx_linux.run`
  - Running NVIDIA samples sometimes fix issues (1\_Uutilities/deviceQuery/)

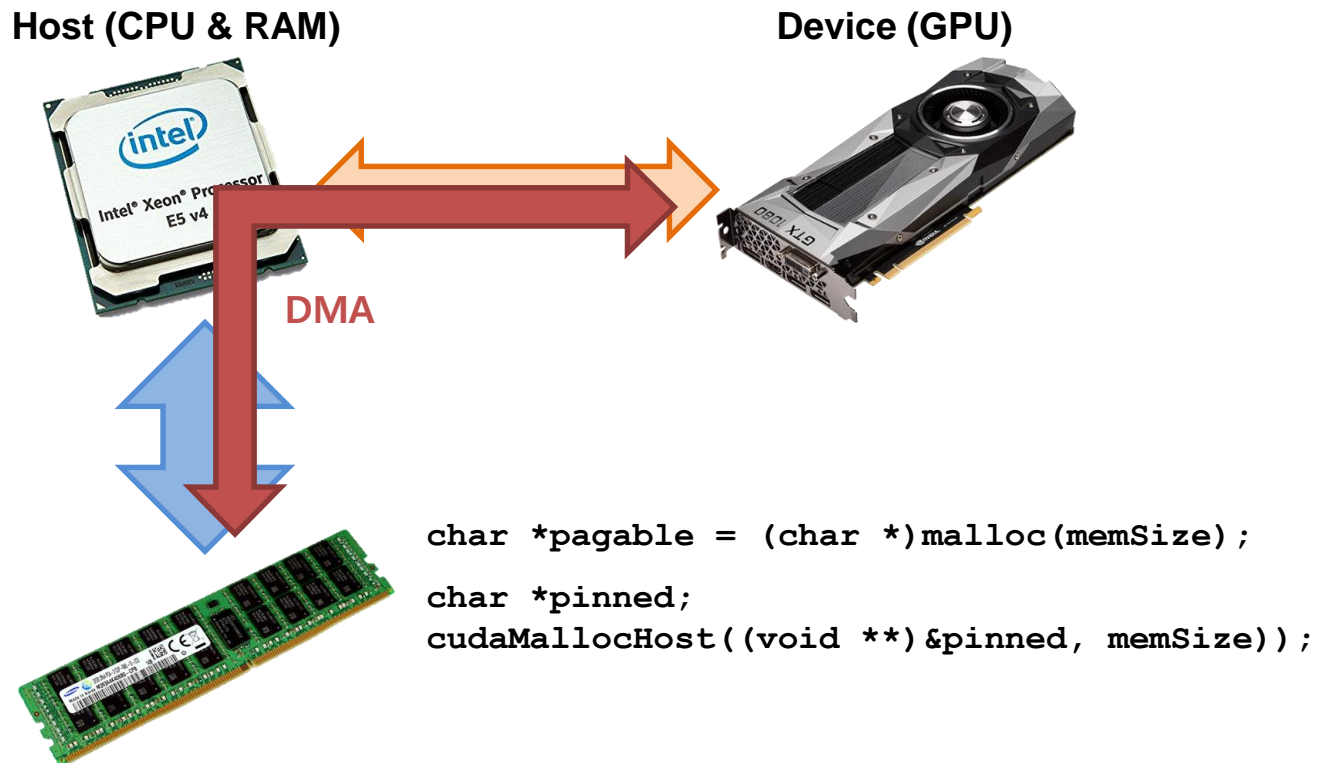
# Heterogeneous Computing – Host and Device

- CPU operates as the *Host* of every NVIDIA GPU *Devices*
- *CUDA* code consists of Host code and Device code
  - *Host* code controls memory and thread execution of a *Device*
  - *Device* code (or *CUDA kernel*) defines operations of GPU threads



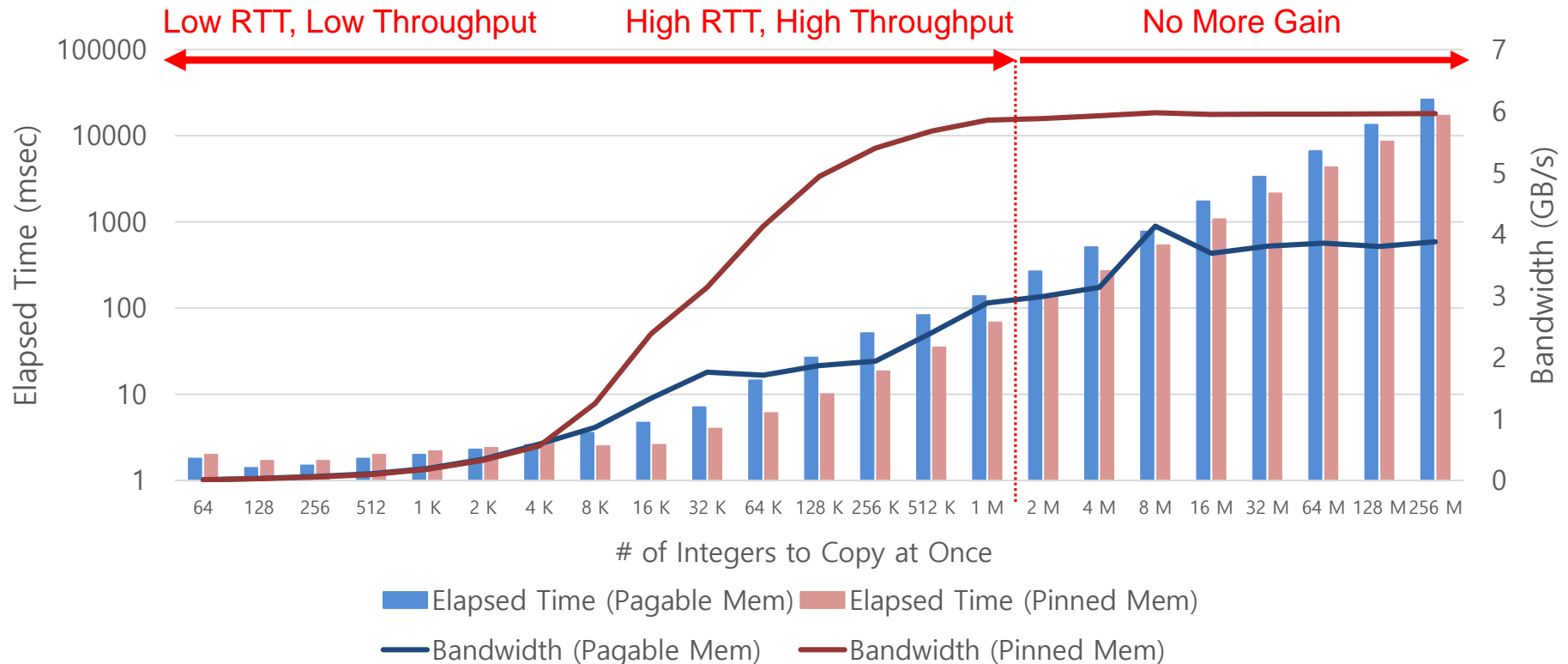
# Heterogeneous Computing – Pinned Memory

- *Pinned (page-locked) memory* is transferred via DMA
  - Frees CPU to work simultaneously during the memory copy
  - Achieves higher bandwidth than pageable memory copy
- Use `cudaMallocHost()` to allocate pinned host memory



# Example: Simple Memory Copy

- Run `make simple_copy` to compile `src/simple_copy/main.cu`
- Run `./simple_copy 64` to copy 64 integers to GPU, launch a null kernel, and copy back to RAM
- The code executes these `nloop=100` times to calculate stable memcpy bandwidth
- Check the elapsed time and bandwidth while increasing the number of integers to copy at once
- Check the difference between using pageable memory and using pinned memory

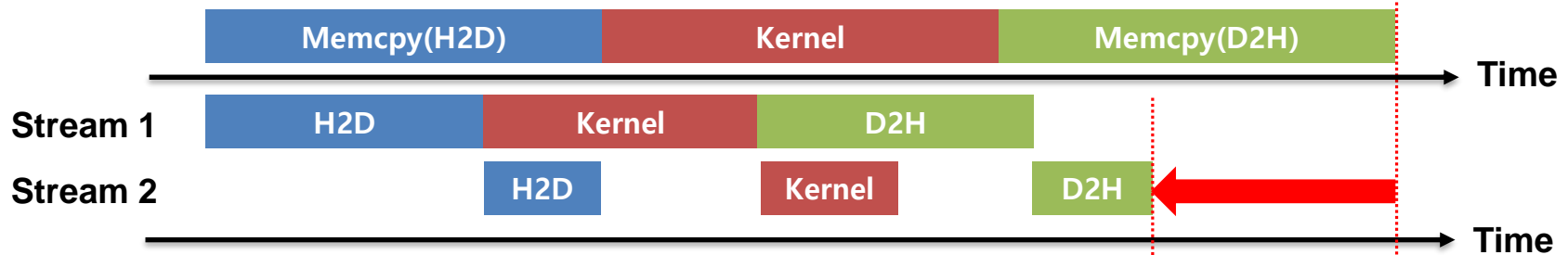


# CUDA Stream in a Device

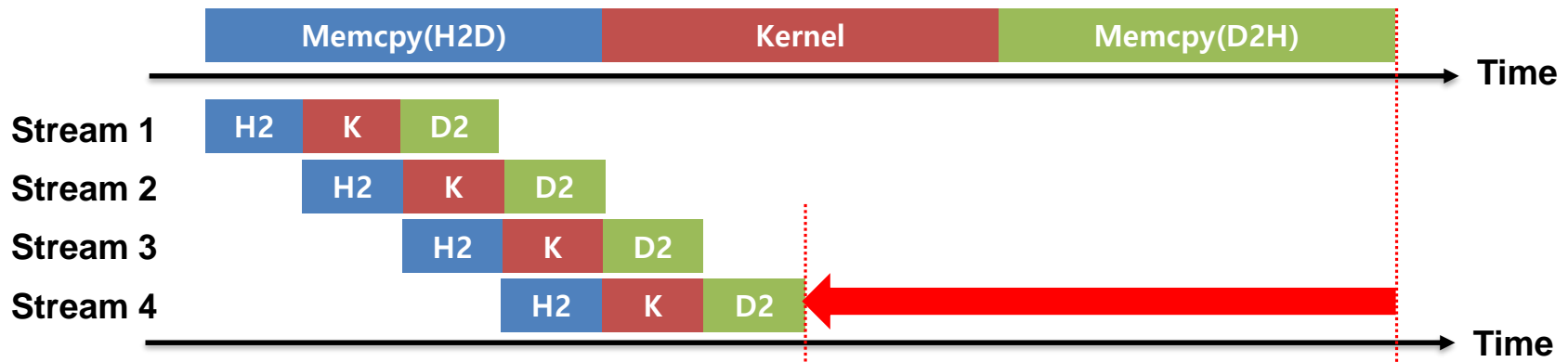
- Stream is a queue of CUDA operations which are issued in order



- Multiple streams issue operations independently, but multiple (H2D copies / Kernels / D2H copies) cannot be executed simultaneously within a device



- We can hide most memory copy overhead (PCIe traffic) via overlapping many streams



# Asynchronous Memory Copy

- Asynchronous memory copy is necessary to use streams
  - Asynchronous: enqueue the job and control returns to CPU immediately
  - c.f. Synchronous: enqueue the job and wait until it finishes, then control returns to CPU
- Examples: copying 16 bytes using synchronous or asynchronous copy

```
cudaMemcpy(device, host, 16, cudaMemcpyHostToDevice);
my_kernel<<<blocks, threads, 0>>>(device, 16);
cudaMemcpy(host, device, 16, cudaMemcpyDeviceToHost);
```

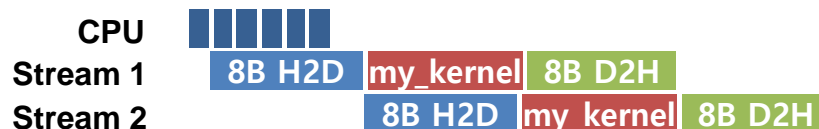


※ kernel launches are always asynchronous

```
cudaMemcpyAsync(device, host, 16, cudaMemcpyHostToDevice, stream1);
my_kernel<<<blocks, threads, 0, stream1>>>(device, 16);
cudaMemcpyAsync(host, device, 16, cudaMemcpyDeviceToHost, stream1);
```



```
cudaMemcpyAsync(device, host, 8, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(device+8, host+8, 8, cudaMemcpyHostToDevice, stream2);
my_kernel<<<blocks, threads, 0, stream1>>>(device, 8);
my_kernel<<<blocks, threads, 0, stream2>>>(device+8, 8);
cudaMemcpyAsync(host, device, 8, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(host+8, device+8, 8, cudaMemcpyDeviceToHost, stream2);
```





# Synchronization – Using Default Stream

- If the stream is not specified, jobs are enqueued in the *default stream* (stream 0)
- Default stream is not overlapped with other streams
- We can use this to operate synchronization throughout all streams
  - Example: running the last synchronizing kernel after all previous asynchronous jobs end

```

cudaMemcpyAsync(device, host, 8, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(device+8, host+8, 8, cudaMemcpyHostToDevice, stream2);
my_kernel<<<blocks, threads, 0, stream1>>>(device, 8);
my_kernel<<<blocks, threads, 0, stream2>>>(device+8, 8);
cudaMemcpyAsync(host, device, 8, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(host+8, device+8, 8, cudaMemcpyDeviceToHost, stream2);
last_kernel<<<blocks, threads, 0>>>(device, 16);
cudaMemcpy(host, device, 16, cudaMemcpyDeviceToHost);

```



# Synchronization – Using CUDA APIs

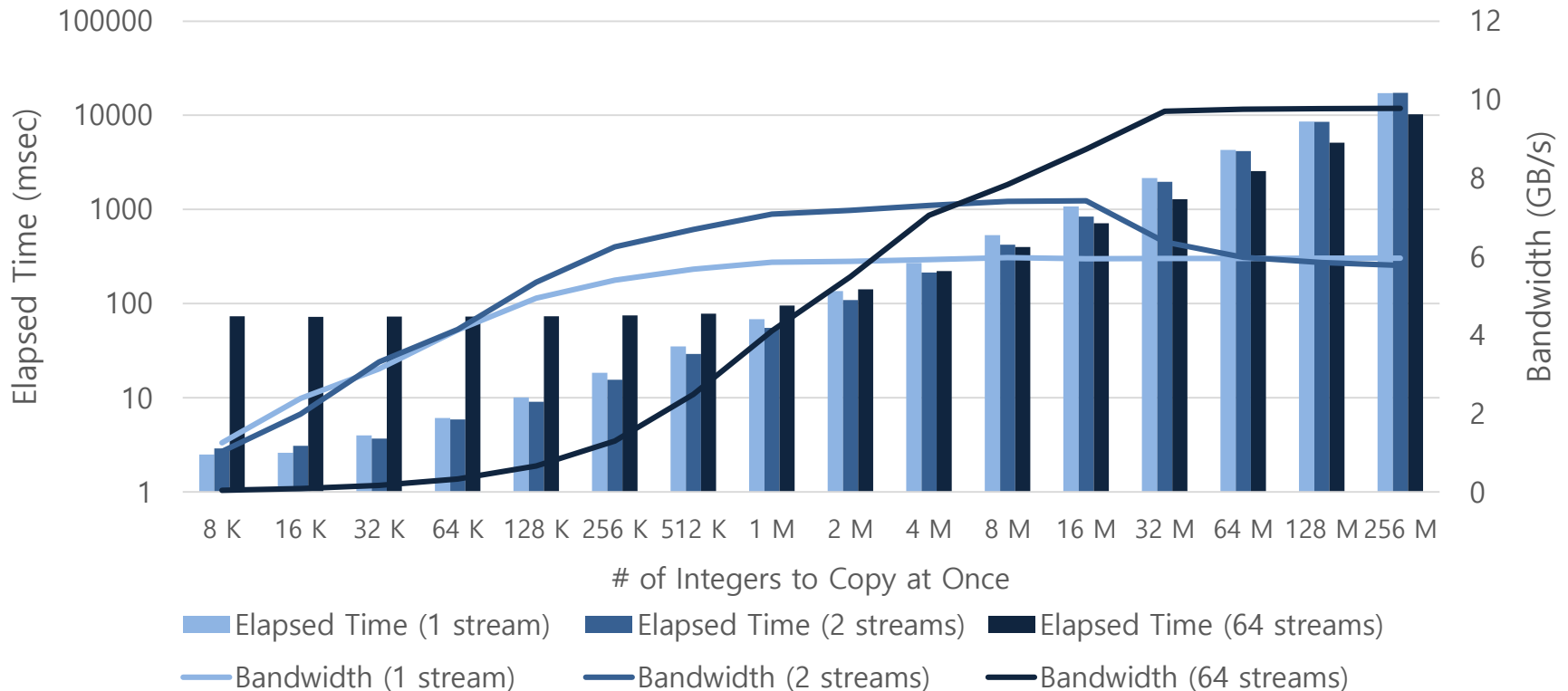
- Some APIs are provided only for synchronization
  - Use `cudaStreamSynchronize()` to synchronize jobs in a stream
  - Use `cudaDeviceSynchronize()` to synchronize every previous jobs in a device
  - Use CUDA events (out of this presentation's scope)

1 <code>cudaMalloc(...);</code>	→	Line 1 finished
2 <code>cudaMemcpyAsync(...H2D...);</code>	→	Line 2 may not have finished
3 <code>my_kernel&lt;&lt;&lt;...&gt;&gt;&gt;(...);</code>	→	Line 2, 3 may not have finished
4 <code>cudaMemcpyAsync(...D2H...);</code>	→	Line 2, 3, 4 may not have finished
5 <code>cudaDeviceSynchronize();</code>	→	Line 2, 3, 4 finished

- Synchronization degrades performance – use it only when it is necessary
- Be careful not to be unaware of using synchronous APIs
  - e.g. `cudaMalloc()` or `cudaFree()` synchronizes the device
  - Check the CUDA API document to be sure whether an API has synchronous properties

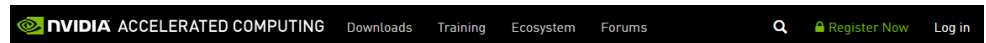
# Example: Memory Copy with Multiple Streams

- Run `make busy_copy` to compile `src/busy_copy/main.cu`
- Similar with `simple_copy` but uses pinned memory only and uses multiple streams
- Run `./busy_copy 1048576 2` to copy 1M integers using 2 streams
- Check the elapsed time and bandwidth while increasing the number of streams
  - The number of streams should be power of 2 for exact comparisons



# CUDA Kernel

- Implementing an optimized CUDA kernel requires huge effort
- We can use provided libraries instead to reduce time
  - e.g. NVIDIA Deep Learning SDK (visit <https://developer.nvidia.com/deep-learning-software>)
- Though we use provided libraries, we need to understand GPU and CUDA to use them properly
- Sometimes we also need to implement a new kernel for untypical operations



Home > ComputeWorks > Deep Learning > Deep Learning Software

## Deep Learning Software

Deep learning algorithms use large amounts of data and the computational power of the GPU to learn information directly from data such as images, signals, and text. NVIDIA® DIGITS offers an interactive workflow-based solution for image classification. Deep learning frameworks offer more flexibility with designing and training custom deep neural networks and provide interfaces to common programming language. The NVIDIA Deep Learning SDK offers powerful tools and libraries for the development of deep learning frameworks such as Caffe, CNTK, TensorFlow, Theano, and Torch.

### NVIDIA Deep Learning SDK

The NVIDIA Deep Learning SDK provides powerful tools and libraries for designing and deploying GPU-accelerated deep learning applications. It includes libraries for deep learning primitives, inference, video analytics, linear algebra, sparse matrices, and multi-GPU communications.



- **Deep Learning Primitives (cuDNN)**: High-performance building blocks for deep neural network applications including convolutions, activation functions, and tensor transformations
- **Deep Learning Inference Engine (TensorRT)**: High-performance deep learning inference runtime for production deployment
- **Deep Learning for Video Analytics (DeepStream SDK)**: High-level C++ API and runtime for GPU-accelerated transcoding and deep learning inference
- **Linear Algebra (cuBLAS)**: GPU-accelerated BLAS functionality that delivers 6x to 17x faster performance than CPU-only BLAS libraries
- **Sparse Matrix Operations (cuSPARSE)**: GPU-accelerated linear algebra subroutines for sparse matrices that deliver up to 8x faster performance than CPU BLAS (MKL), ideal for applications such as natural language processing
- **Multi-GPU Communication (NCCL)**: Collective communication routines, such as all-gather, reduce, and broadcast that accelerate multi-GPU deep learning training on up to eight GPUs

# CUDA Kernel Programming Architecture

- GPU computation cores are divided in several *streaming multiprocessors* (SMs or MPs)
  - # of SMs depends on devices
- *Warp*: a set of 32 threads
  - The scheduling unit of a SM → threads in a warp are always synchronized with each other!
  - Supports *shuffle instructions* between threads in a warp, such as `__shfl_up()`, `__shfl_xor()`, etc.
  - # of warps per SM depends on *CUDA compute capability*
  - Maximum # of resident threads = (# of SMs) \* (# of warps per SM) \* (32 threads per warp)
    - e.g. GTX 1080: (20 SMs) \* (4 warps per SM) \* (32 threads per warp) = 2560 resident threads
- *Block*: a set of one or more threads
  - Maximum # of threads per block depends on devices
    - e.g. GTX 1080: 1024 threads per block
  - Threads in a block shares a specified amount of *shared memory* (faster access than global memory)
  - Threads in a block can be synchronized via `__syncthreads()`
- *Grid*: a set of one or more blocks
  - All blocks are basically independent, no API for synchronization
  - We can use some signaling APIs between blocks instead, such as memory fence or atomic operations

# Kernel Implementation Tips – 1

- Don't branch the context of the kernel
  - Use if-else as less as possible
  - Make the condition constant during the runtime
    - Comparison between constants are calculated during compilation
    - Use C++ template to make some user-decided variables constant
  - Branching inside a warp makes the performance terrible
- Memory access optimization
  - Sequential access is much faster than random access
  - Remember that memory access is much slower than calculations
  - Read data as much as possible at once
  - Let frequently accessed data be in shared memory or local memory
- Warp optimization
  - Make the best use of shuffle instructions
  - Never branch in a warp
  - Always let threads in a warp be in the same block
- Synchronization
  - Overhead Comparison: Device Sync > Stream Sync >> Using Event > Block Sync
  - Utilize features of warps or shared memory to reduce synchronization

# Kernel Implementation Tips – 2

- Deciding GPU resource usage
  - Always compute the maximum # of resident threads in the code and store it as a constant
  - Decide the # of threads per block according to your usage plan of shared memory
    - Usually 256, 512, or 1024 is used
  - Always let the # of threads per block power of 2 and larger than or equal to 32
    - 32, 64, 128, 256, 512, or 1024
  - Don't let the # of threads in a grid exceed the maximum # of resident threads
  - If there are many kinds of kernels to be executed in parallel, hand out resident threads according to workload amount
  - If kernels to be executed are serial, provide the whole resident threads to every kernel executions
- Profiling
  - Find out which kernels are slow via measuring pure kernel latency without memory copy
  - Measure with different size of workload to know in which state the kernel performs best
  - Profiling tools like `nvprof` and `nvvp` may reduce profiling effort

# Example: Reduction

- Run `make reduction` to compile `src/reduction/main.cu` and `reduction.cuh`
- Run `./reduction 1024` to test adding 1024 integers
- Reading and understanding `reduction.cuh` may be very helpful to understand basic kernel programming



# Useful References

- CUDA streams: best practices and common pitfalls  
<http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf>
- CUDA C/C++ Streams and Concurrency  
<http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf>
- Optimizing Parallel Reduction in CUDA  
[http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf)
- Parallel Prefix Sum (Scan) with CUDA  
[http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch39.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html)

## Thank You.