Parallel Computing 2018 Spring

中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

(1) Getting Started

References:

1. An Even Easier Introduction to CUDA. By Mark Harris | January 25, 2017. https://devblogs.nvidia.com/even-easier-introduction-cuda/.

2. Unified Memory for CUDA Beginners. By Mark Harris | June 19, 2017, https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

3. NVDIA Developer Blog. https://devblogs.nvidia.com/

1. **CUDA Overview**
2. An Even Easier Introduction to CUDA
3. Unified Memory for CUDA Beginners

What is CUDA

- CUDA C++ is just one of the ways you can create massively parallel applications with CUDA.
- It lets you use the powerful C++ programming language to develop high performance algorithms accelerated by thousands of parallel threads running on GPUs.
- Many developers have accelerated their computation- and bandwidth-hungry applications this way, including the libraries and frameworks that underpin the ongoing revolution in artificial intelligence known as Deep Learning.

Requirements

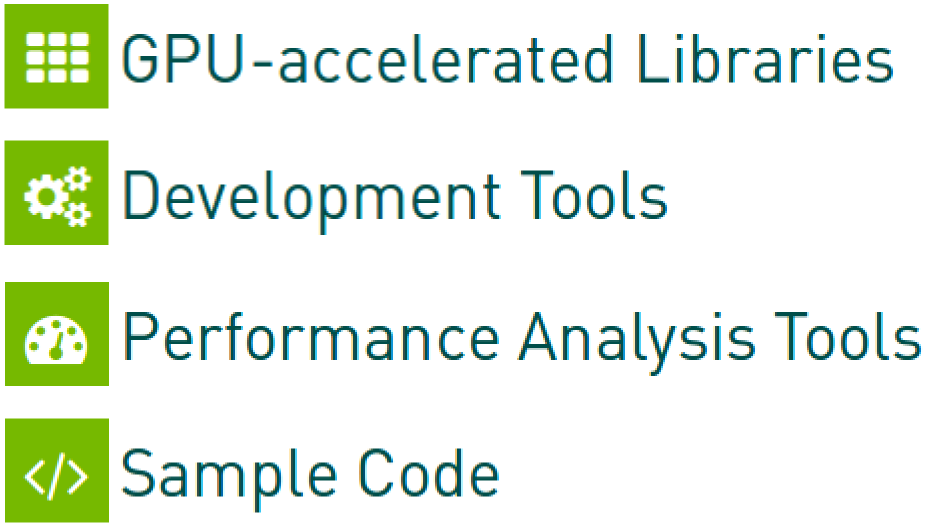
- C/C++
- Computer with an CUDA-capable GPU
  - Windows, Mac, or Linux,
  - NVIDIA GPUs
  - Cloud instance with GPUs (AWS, Azure, IBM SoftLayer...)
- **... CUDA**

CUDA Toolkit

- The NVIDIA® CUDA® Toolkit provides a development environment for creating high performance GPU-accelerated applications.
- Develop, optimize and deploy your applications on:
  - GPU-accelerated embedded systems / desktop workstations
  - enterprise data centers / cloud-based platforms
  - HPC supercomputers.
- Enable drop-in acceleration across multiple domains:
  - linear algebra
  - image and video processing
  - deep learning
  - graph analytics

CUDA Toolkit

- Components



GPU-accelerated Libraries

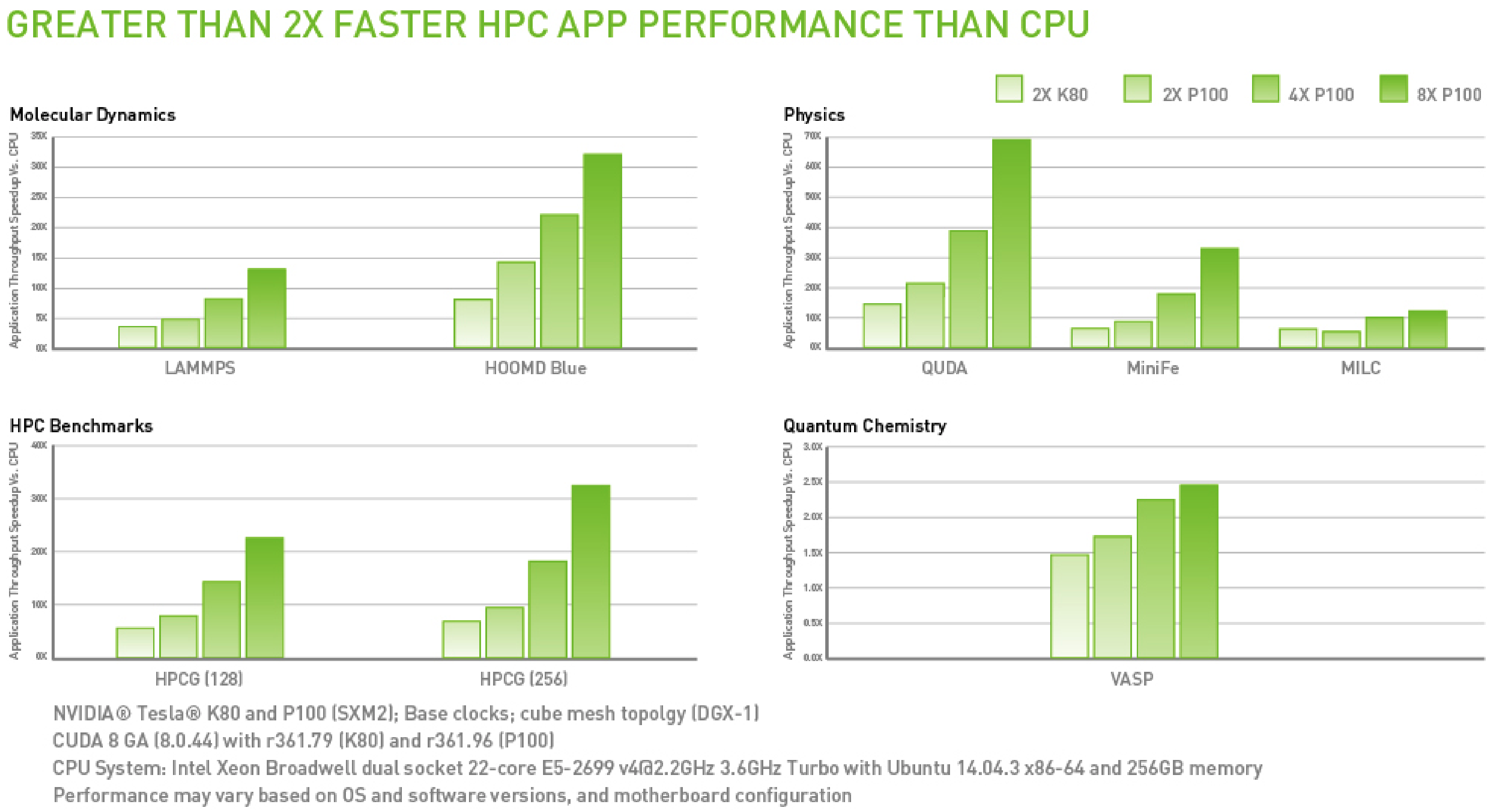
Development Tools

Performance Analysis Tools

Sample Code

CUDA Toolkit

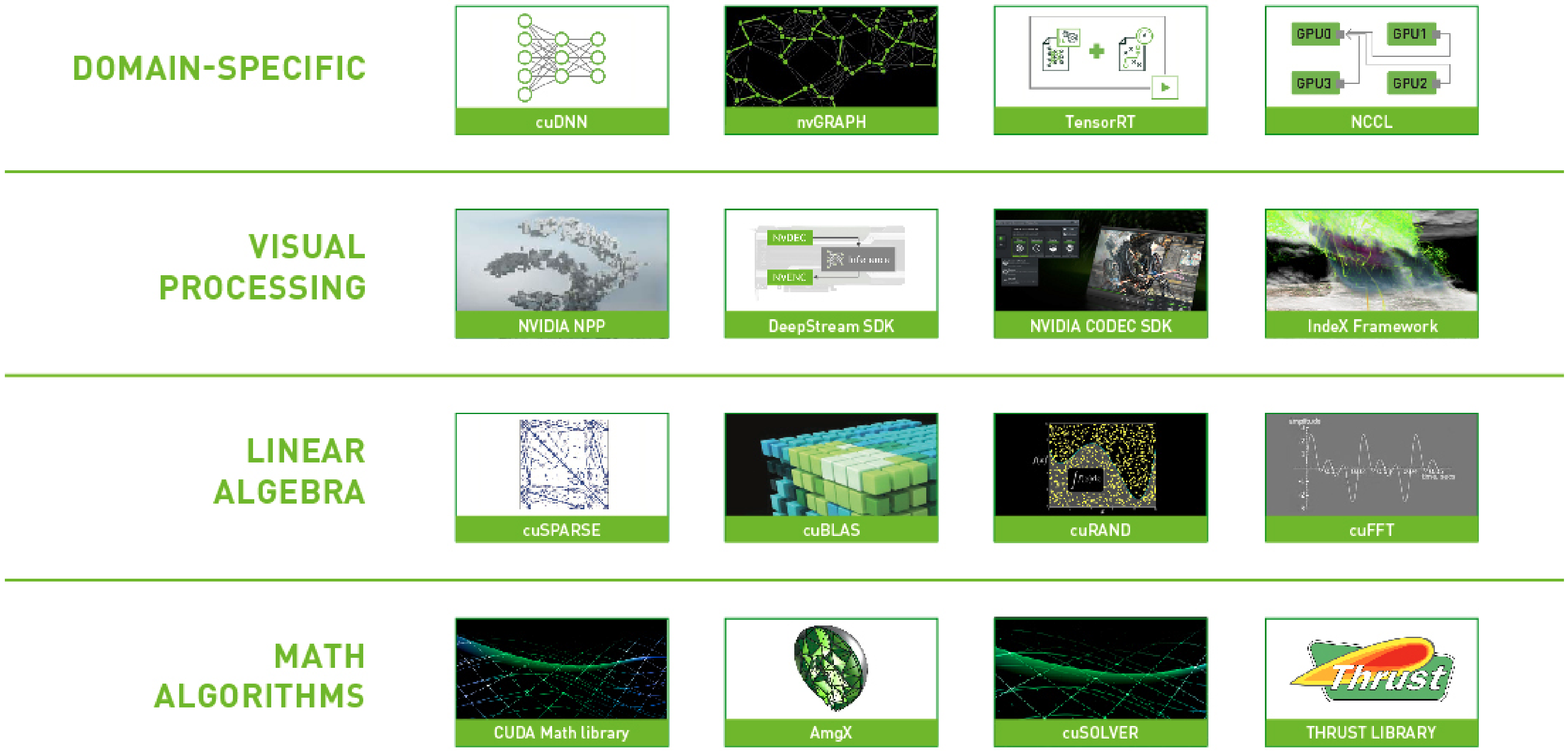
- High Performance



NVIDIA® Tesla® K80 and P100 (SXM2); Base clocks; cube mesh topolgy (DGX-1)
CUDA 8 GA (8.0.44) with r361.79 (K80) and r361.96 (P100)
CPU System: Intel Xeon Broadwell dual socket 22-core E5-2699 v4@2.2GHz 3.6GHz Turbo with Ubuntu 14.04.3 x86-64 and 256GB memory
Performance may vary based on OS and software versions, and motherboard configuration

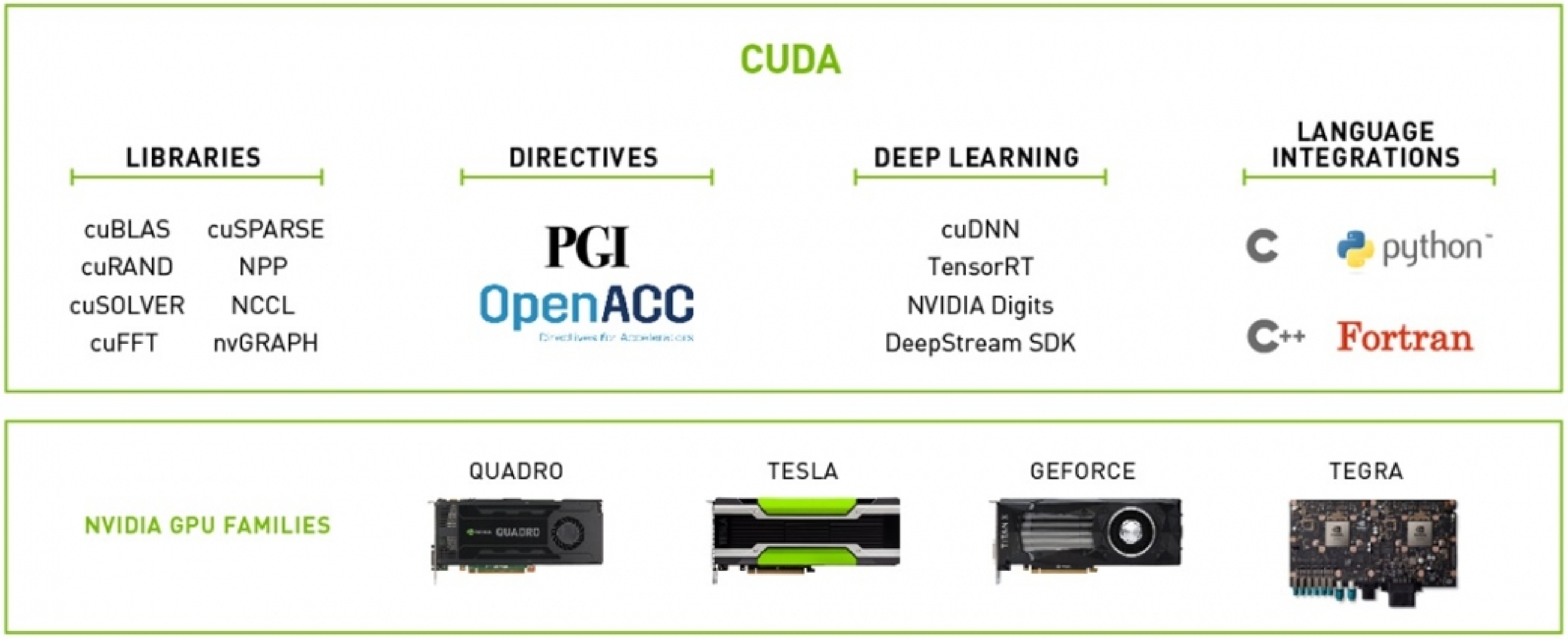
CUDA Toolkit

- Domain Specific Libraries



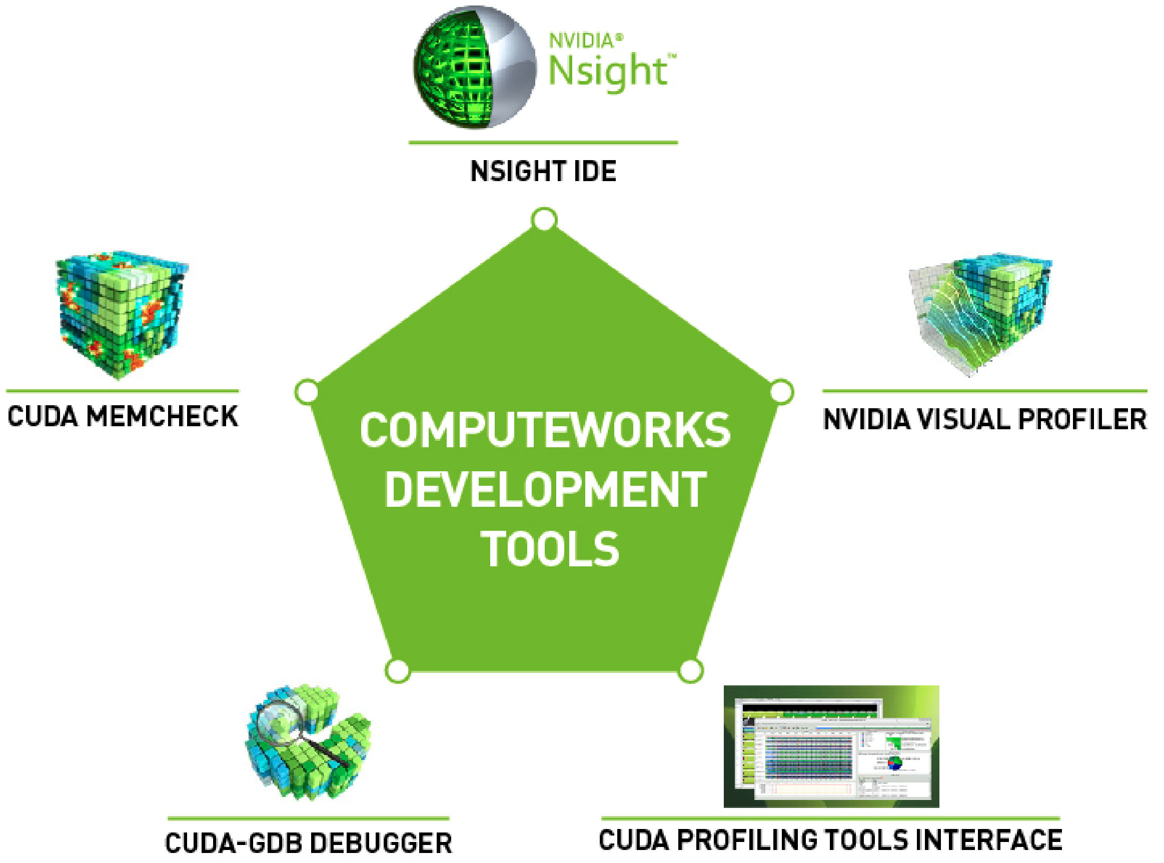
CUDA Toolkit

- Develop Once, Deploy Everywhere



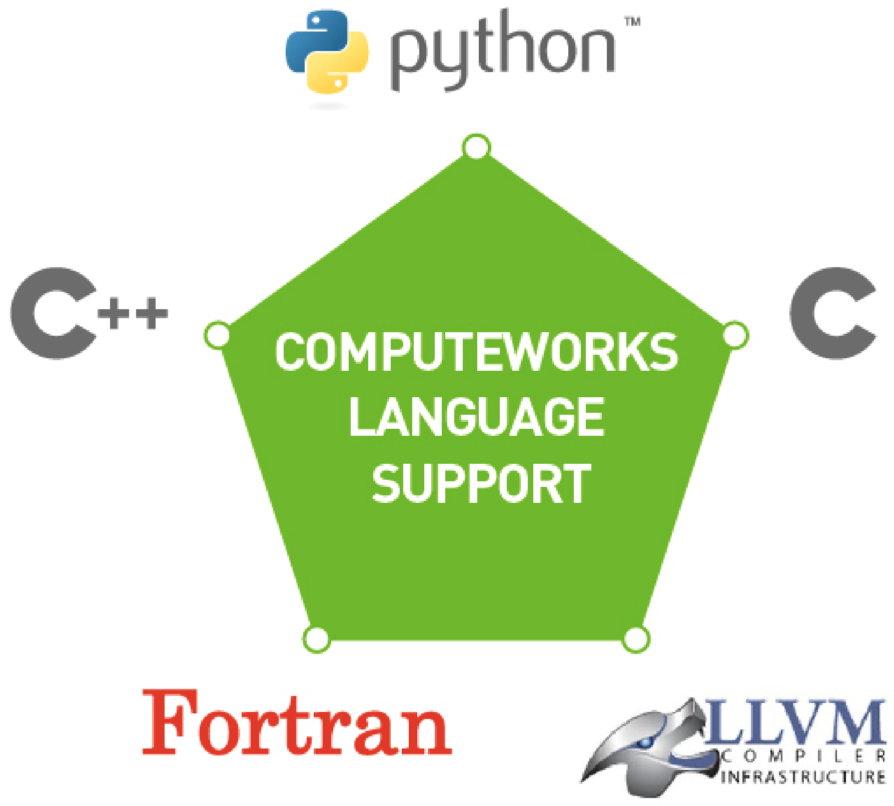
CUDA Toolkit

- Integrated Development Environment



CUDA Toolkit

- Language Integration



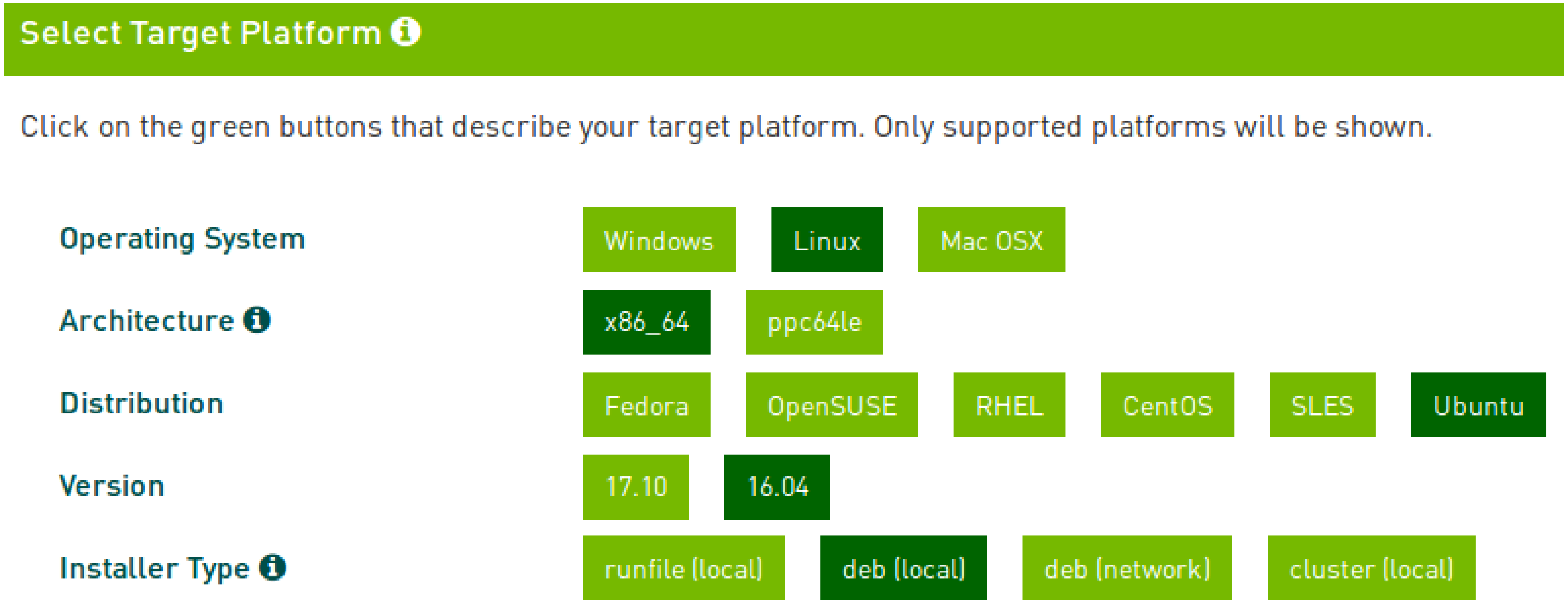
CUDA Toolkit

- Built-in integration

Adobe ANSYS AUTODESK DASSAULT SYSTEMES

MathWorks Microsoft NATIONAL INSTRUMENTS Wolfram Mathematica

CUDA Download



Select Target Platform

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

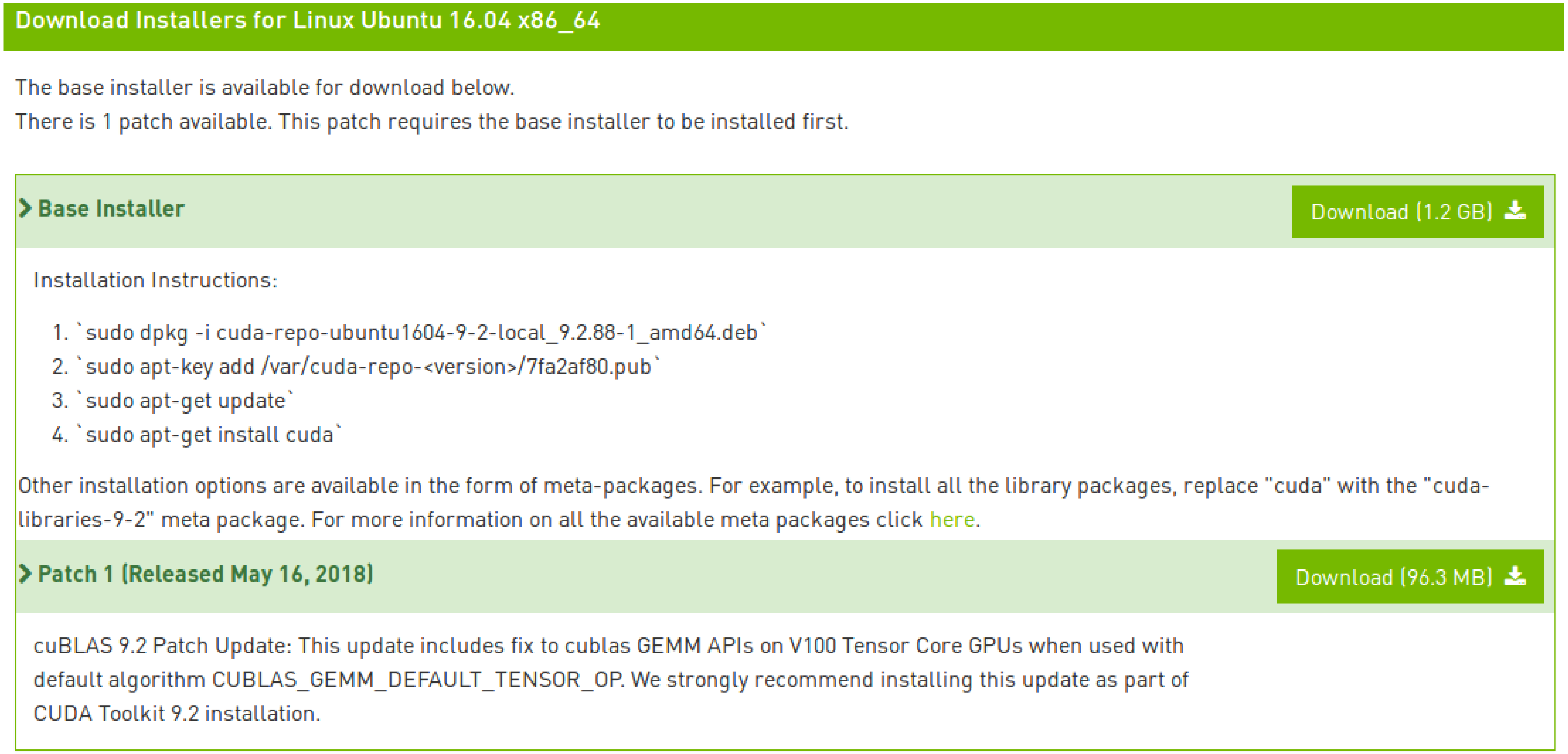
Operating System	Windows [✓ Linux] Mac OSX
Architecture	[✓ x86_64] ppc64le
Distribution	Fedora OpenSUSE RHEL CentOS SLES [✓ Ubuntu]
Version	17.10 [✓ 16.04]
Installer Type	runfile (local) [✓ deb (local)] deb (network) cluster (local)

CUDA Pre-Installation

- Verify You Have a CUDA-Capable GPU
  - $ lspci | grep -i nvidia
- Verify You Have a Supported Version of Linux
  - $ uname -m && cat /etc/*release
- Verify the System Has gcc Installed
  - $ gcc --version
- Verify the System has the Correct Kernel Headers and Development Packages Installed
  - $ sudo apt-get install linux-headers-$(uname -r)

NVIDIA CUDA Installation Guide for Linux.
https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

CUDA Installation



Download Installers for Linux Ubuntu 16.04 x86_64

The base installer is available for download below.
There is 1 patch available. This patch requires the base installer to be installed first.

> Base Installer — Download (1.2 GB)

Installation Instructions:

1. `sudo dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb`
2. `sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub`
3. `sudo apt-get update`
4. `sudo apt-get install cuda`

Other installation options are available in the form of meta-packages. For example, to install all the library packages, replace "cuda" with the "cuda-libraries-9-2" meta package. For more information on all the available meta packages click here.

> Patch 1 (Released May 16, 2018) — Download (96.3 MB)

cuBLAS 9.2 Patch Update: This update includes fix to cublas GEMM APIs on V100 Tensor Core GPUs when used with default algorithm CUBLAS_GEMM_DEFAULT_TENSOR_OP. We strongly recommend installing this update as part of CUDA Toolkit 9.2 installation.

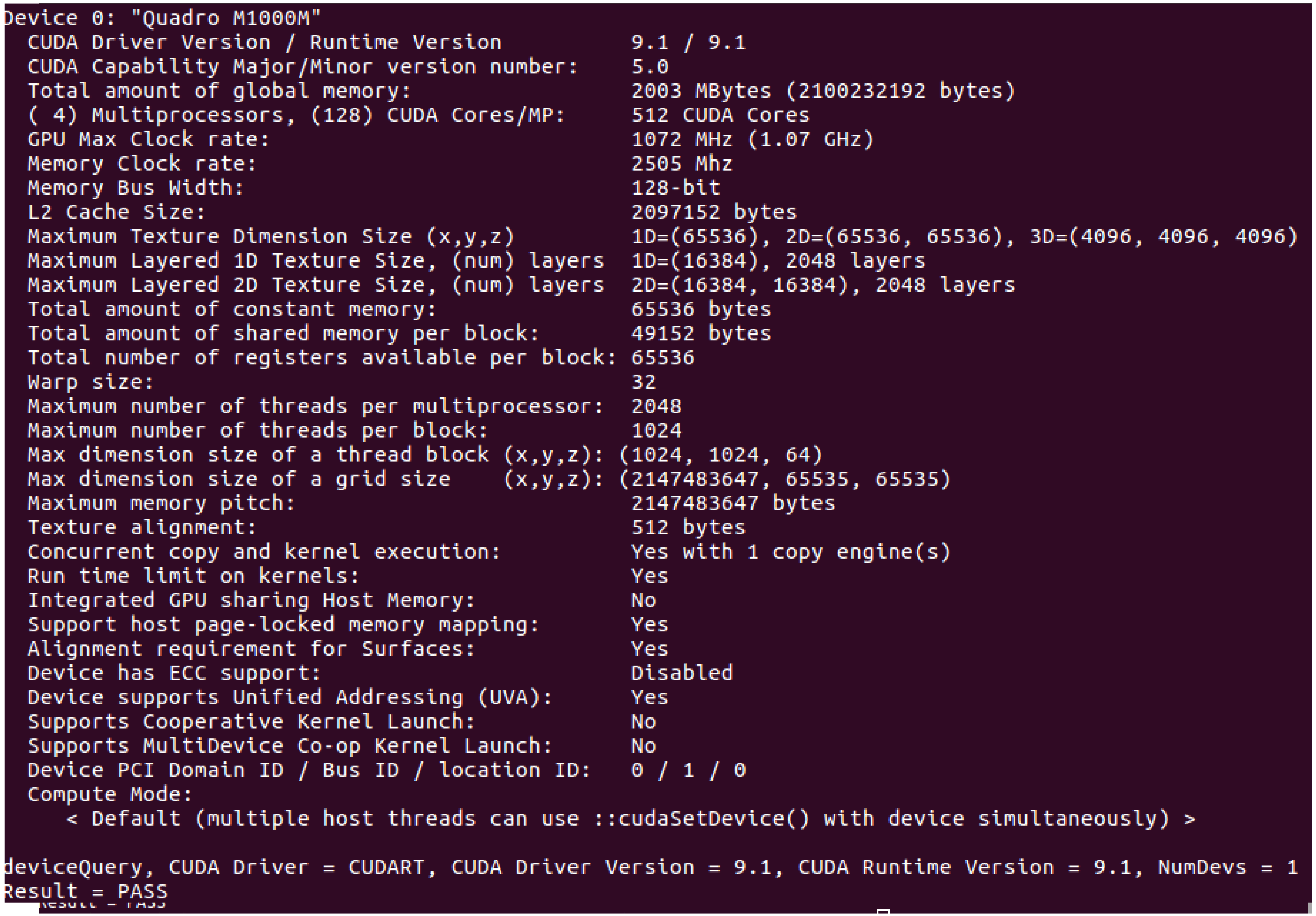
1. CUDA Quick Start Guide, DU-05347-301_v9.1 | March 2018.
2. NVIDIA CUDA Installation Guide for Linux. https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

CUDA Post-Installation

- Environment Setup
  - export PATH=/usr/local/cuda-9.2/bin${PATH:+:${PATH}}
  - export LD_LIBRARY_PATH=/usr/local/cuda-9.2/lib64\${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
-
- Install Writable Samples
  - $ cuda-install-samples-9.2.sh <dir>
  - $ make

CUDA Verification

- Verify the Driver Version
  - $ cat /proc/driver/nvidia/version
- Compiling the Examples
  - Go to ~/NVIDIA_CUDA-9.2_Samples and typing make.
  - The resulting binaries will be placed under ~/NVIDIA_CUDA-9.2_Samples/bin.
- Running the Binaries
  - under ~/NVIDIA_CUDA-9.2_Samples
  - Find and run **deviceQuery**.



```
Device 0: "Quadro M1000M"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2003 MBytes (2100232192 bytes)
  ( 4) Multiprocessors, (128) CUDA Cores/MP:     512 CUDA Cores
  GPU Max Clock rate:                            1072 MHz (1.07 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.1, CUDA Runtime Version = 9.1, NumDevs = 1
Result = PASS
```

1. CUDA Overview

2. **An Even Easier Introduction to CUDA**

3. Unified Memory for CUDA Beginners

Starting Simple

- A simple C++ program that adds the elements of two arrays with a million elements each:
  - add.cpp
  - Compile with a C++ complier
    - clang++ add.cpp -o add
    - g++ add.cpp -o add
  - Run it
    - ./add

Running in parallel

- Step1:
- Turn add function into a *kernel*
  - Kernel:
    - A function that runs on the GPU and can be called from CPU code
  - __global__
    - tells the CUDA C++ compiler that this is a kernel

Running in parallel

- Step1:
- Turn add function into a *kernel*

```
//CUDA Kernel function to add elements of two arrays on GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

These __global__ functions are known as *kernels;*

Code that runs on the GPU is often called *device* code,
while code that runs on the CPU is *host* code.

Running in parallel

- Step2:
- Memory Allocation in CUDA
  - To compute on the GPU, you need to allocate memory accessible by the GPU.
  - Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system.
  - To allocate data in unified memory, call *cudaMallocManaged(),* which returns a pointer that you can access from host (CPU) code or device (GPU) code.
    - new → cudaMallocManaged()
    - delete → cudaFree

Running in parallel

- Step2:
- Memory Allocation in CUDA

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

Running in parallel

- Step3:
- Launch the kernel
  - CUDA kernel launches are specified using the triple angle bracket syntax <<< >>>

```
add<<<1, 1>>>(N, x, y);
```

This line launches one GPU thread to run add().

Running in parallel

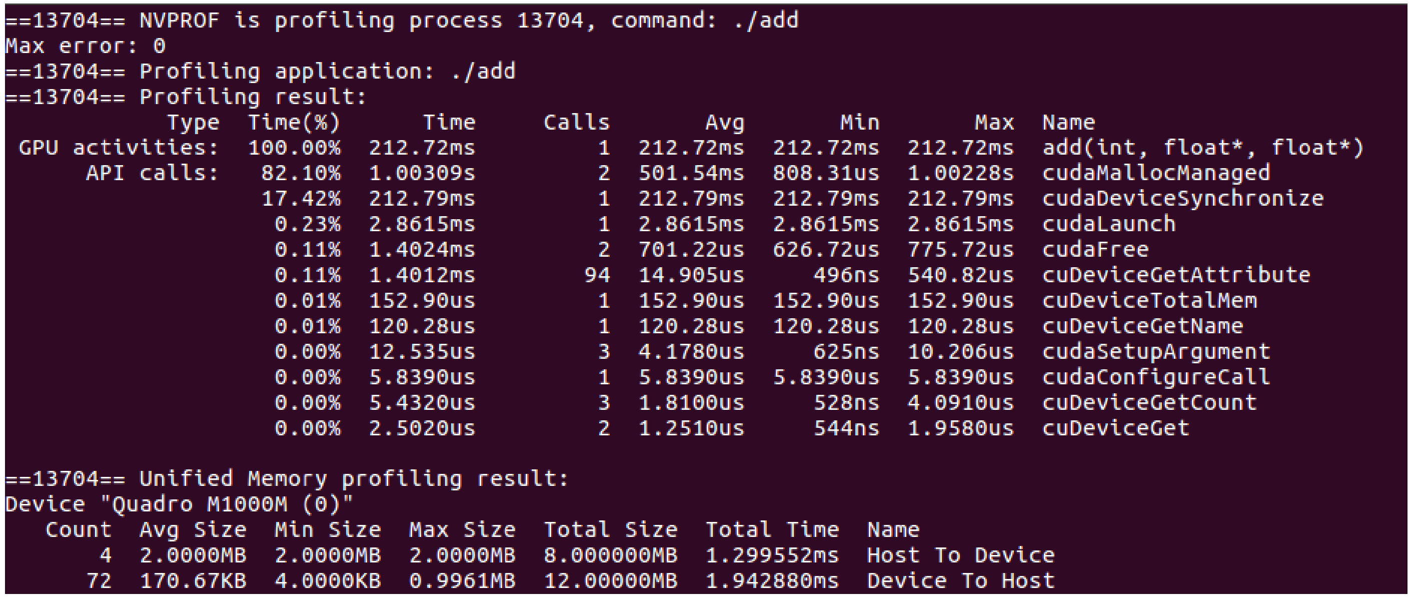
- One more thing:
- Synchronization
  - CPU needs to wait until the kernel is done before it accesses the results;
  - CUDA kernel launches don’t block the calling CPU thread;
  - Just call cudaDeviceSynchronize() before doing the final error checking on the CPU.

Running in parallel

- Compile and Run:
  - Save file
    - Change *add.cpp* to *add.cu*
  - Compile with nvcc the CUDA C++ compiler
    - nvcc add.cu -o add_cuda
  - Run
    - ./add_cuda

Running in parallel

- Profile it:
  - nvprof
    - the simplest way to find out how long the kernel takes to run
    - the command line GPU profiler that comes with the CUDA Toolkit.
    - *nvprof ./add_cuda*



```
==13704== NVPROF is profiling process 13704, command: ./add
Max error: 0
==13704== Profiling application: ./add
==13704== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  212.72ms         1  212.72ms  212.72ms  212.72ms  add(int, float*, float*)
      API calls:   82.10%  1.00309s         2  501.54ms  808.31us  1.00228s  cudaMallocManaged
                   17.42%  212.79ms         1  212.79ms  212.79ms  212.79ms  cudaDeviceSynchronize
                    0.23%  2.8615ms         1  2.8615ms  2.8615ms  2.8615ms  cudaLaunch
                    0.11%  1.4024ms         2  701.22us  626.72us  775.72us  cudaFree
                    0.11%  1.4012ms        94  14.905us     496ns  540.82us  cuDeviceGetAttribute
                    0.01%  152.90us         1  152.90us  152.90us  152.90us  cuDeviceTotalMem
                    0.01%  120.28us         1  120.28us  120.28us  120.28us  cuDeviceGetName
                    0.00%  12.535us         3  4.1780us     625ns  10.206us  cudaSetupArgument
                    0.00%  5.8390us         1  5.8390us  5.8390us  5.8390us  cudaConfigureCall
                    0.00%  5.4320us         3  1.8100us     528ns  4.0910us  cuDeviceGetCount
                    0.00%  2.5020us         2  1.2510us     544ns  1.9580us  cuDeviceGet

==13704== Unified Memory profiling result:
Device "Quadro M1000M (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB  1.299552ms  Host To Device
      72  170.67KB  4.0000KB  0.9961MB  12.00000MB  1.942880ms  Device To Host
```

Running in parallel

- Analysis:
  - This is only a first step.
  - The kernel is only correct for a single thread, since every thread that runs it will perform the add on the whole array.
  - Moreover, there is a race condition since multiple parallel threads would both read and write the same locations.

Picking up the Threads

- The key is in CUDA’s <<<1, 1>>> syntax:
  - This is called the *execution configuration*,
  - and it tells the CUDA runtime *how many parallel threads* to use for the launch on the GPU.
- There are two parameters here:
  - but let’s start by changing the second one: *the number of threads in a thread block*.
  - CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose:

```
add<<<1, 256>>>(N, x, y);
```

Picking up the Threads

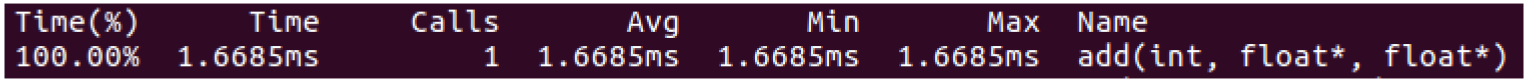
- add<<<1, 256>>> (N, x, y):
  - it will do the computation once per thread,
  - rather than spreading the computation across parallel threads.
- Modify the kernel:
  - CUDA C++ provides keywords that let kernels get the indices of the running threads.
  - ***threadIdx.x***
    - contains the index of the current thread within its block
  - ***blockDim.x***
    - contains the number of threads in the block
  -

Picking up the Threads

- Modify the kernel:
  - ***threadIdx.x***
    - contains the index of the current thread within its block
  - ***blockDim.x***
    - contains the number of threads in the block

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Picking up the Threads

- Modify the kernel:
  - ***threadIdx.x blockDim.x***
  - *In fact, setting index to 0 and stride to 1 makes it semantically identical to the first version.*
- Profile again:
  - *Save the file as **add_block.cu***
  - *compile and run it in nvprof again:*
  - 

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  1.6685ms         1  1.6685ms  1.6685ms  1.6685ms  add(int, float*, float*)
```

  - *big speedup of ~126 with 256 threads.*

Getting even faster

- CUDA GPUs have many parallel processors grouped into Streaming Multiprocessors, or SMs.
- Each SM can run multiple concurrent thread blocks.
  - Tesla P100 GPU (Pascal Architecture)
  - 56 SMs, each capable of supporting up to 2048 active threads.
- To take full advantage of all these threads, I should launch the kernel with multiple thread blocks.

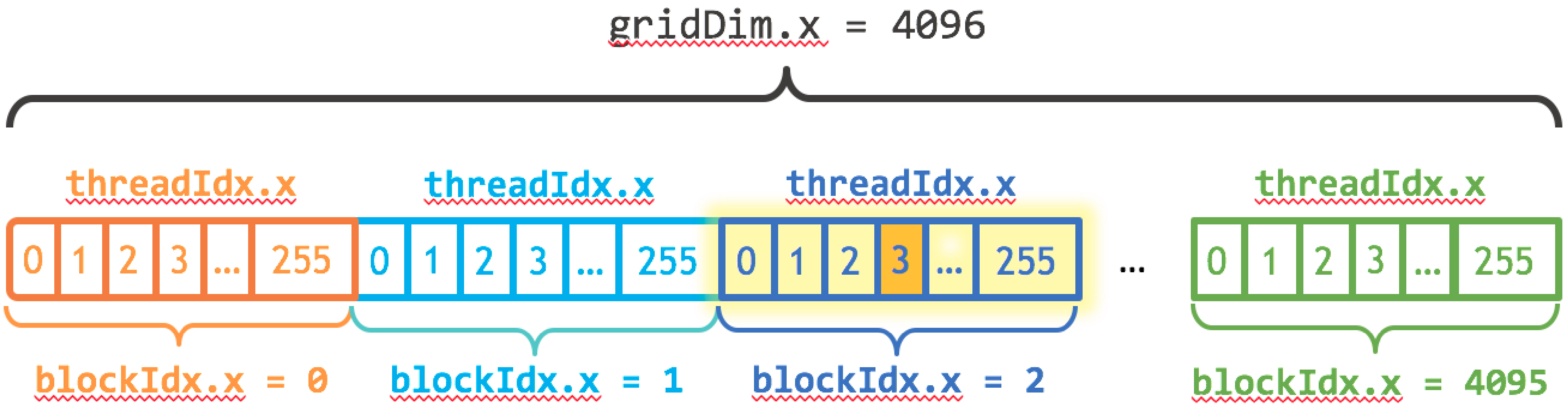
Getting even faster

- The key is in CUDA’s <<<1, 1>>> syntax:
  - the first parameter of the execution configuration specifies the number of thread ***blocks***.
  - Together, the blocks of parallel threads make up the ***grid***.
- Since you have N elements to process, and 256 threads per block, you just need to calculate the number of blocks to get at least N threads.
  - simply divide N by the block size

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

Getting even faster

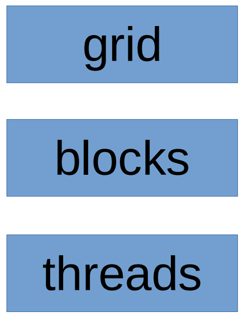
- The key is in `<<<numBlocks, blockSize>>>` syntax:



```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =     (2)     *    (256)    +     (3)     = 515
```

Getting even faster

- The CUDA parallel thread hierarchy:
  - CUDA executes kernels using a grid of blocks of threads.
- Common indexing pattern used in CUDA programs using the CUDA keywords:
  - **gridDim.x**
    - the number of thread blocks
  - **blockDim.x**
    - the number of threads in each block
  - **blockIdx.x**
    - the index the current block within the grid
  - **threadIdx.x**
    - the index of the current thread within the block

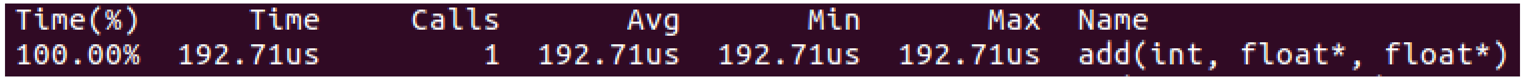


Getting even faster

- Update the kernel code:
  - Gets thread index by:
    - computing the offset to the beginning of its block (the block index times the block size: blockIdx.x * blockDim.x) and adding the thread's index within the block (threadIdx.x).
  - Set thread stride to:
    - the total number of threads in the grid (blockDim.x * gridDim.x).
    - This type of loop in a CUDA kernel is often called a grid-stride loop.

-
  ```
  __global__
  void add(int n, float *x, float *y)
  {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
  }
  ```

Getting even faster

- Update the kernel code:
  - Save the file as **add_grid.cu**
  - Compile and run it in nvprof again
  - 

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  192.71us         1  192.71us  192.71us  192.71us  add(int, float*, float*)
```

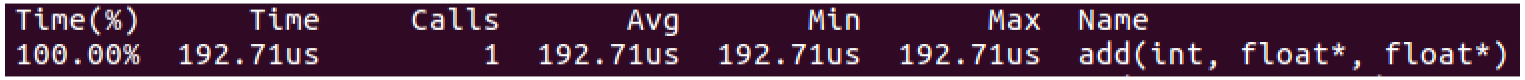
  - Additional ~8.6 speedup.

Excercises

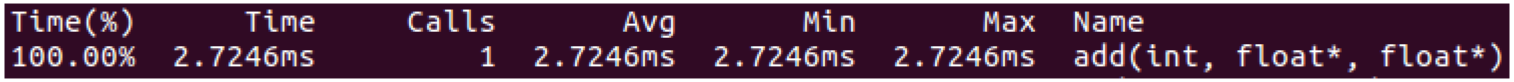
- Experiment with printf() inside the kernel.
- Try printing out the values of threadIdx.x and blockIdx.x for some or all of the threads.
- Do they print in sequential order? Why or why not?

1. CUDA Overview
2. An Even Easier Introduction to CUDA
3. **Unified Memory for CUDA Beginners**

Running on Pascal GPU

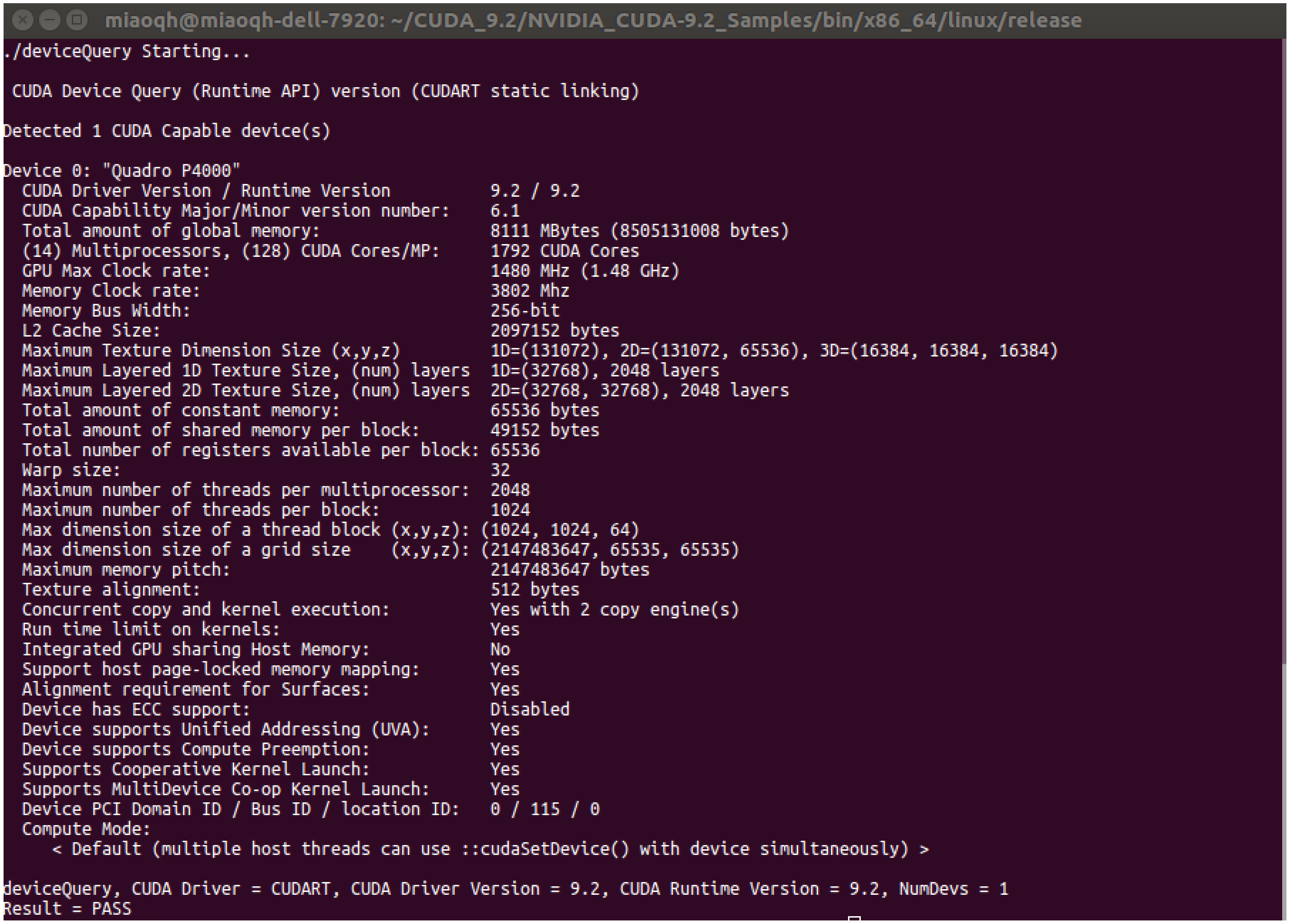
- We had run add_grid.cu on Quadro M1000M:
  - Maxwell architecture
  - RMB~700
  - 

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  192.71us         1  192.71us  192.71us  192.71us  add(int, float*, float*)
```

- Now run add_grid.cu on Quadro P4000:
  - Pascal architecture
  - RMB~7000
  - 

```
Time(%)      Time     Calls       Avg       Min       Max  Name
100.00%  2.7246ms         1  2.7246ms  2.7246ms  2.7246ms  add(int, float*, float*)
```

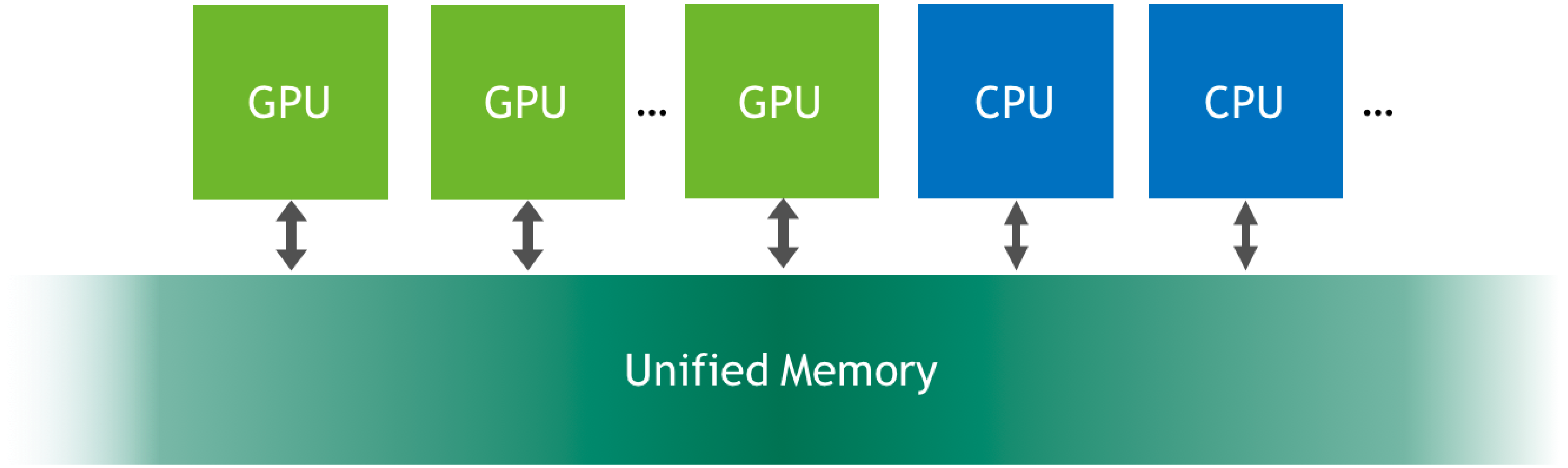
?



```
miaoqh@miaoqh-dell-7920: ~/CUDA_9.2/NVIDIA_CUDA-9.2_Samples/bin/x86_64/linux/release
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro P4000"
  CUDA Driver Version / Runtime Version          9.2 / 9.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8111 MBytes (8505131008 bytes)
  (14) Multiprocessors, (128) CUDA Cores/MP:     1792 CUDA Cores
  GPU Max Clock rate:                            1480 MHz (1.48 GHz)
  Memory Clock rate:                             3802 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 115 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.2, CUDA Runtime Version = 9.2, NumDevs = 1
Result = PASS
```

What is Unified Memory

- Unified Memory is a single memory address space accessible from any processor in a system.



What is Unified Memory

- Unified Memory:
  - This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs.
  - Allocating Unified Memory is as simple as replacing calls to malloc() or new with calls to cudaMallocManaged()
  - returns a pointer accessible from any processor (ptr in the following).

-
  ```
  cudaError_t cudaMallocManaged(void** ptr, size_t size);
  ```

What is Unified Memory

- Unified Memory:
  - `cudaError_t cudaMallocManaged(void** ptr, size_t size);`
  - When code running on a CPU or GPU accesses data allocated this way, the CUDA system *software and/or the hardware* takes care of *migrating memory pages* to the memory of the accessing processor.
- The important point here is that:
  - Older GPUs based on the Kepler and Maxwell architectures support a more limited form of Unified Memory.
  - The ***Pascal*** *GPU architecture is the* ***first*** *with* ***hardware*** support for virtual memory page faulting and migration, via its Page Migration Engine.

What Happens on Kepler/Maxwell

- On systems with pre-Pascal GPUs:
  - calling cudaMallocManaged() allocates size bytes of managed memory on the GPU device that is active when the call is made.
  - Internally, the driver also sets up page table entries for all pages covered by the allocation, so that the system knows that the pages are resident on that GPU.
- So, in our example:
  - x and y are both initially fully resident in GPU memory. Then in the loop starting on line 6, the CPU steps through both arrays, initializing their elements to 1.0f and 2.0f, respectively.
  - Since the pages are initially resident in device memory, a page fault occurs on the CPU for each array page to which it writes, and the GPU driver migrates the page from device memory to CPU memory. After the loop, all pages of the two arrays are resident in CPU memory.

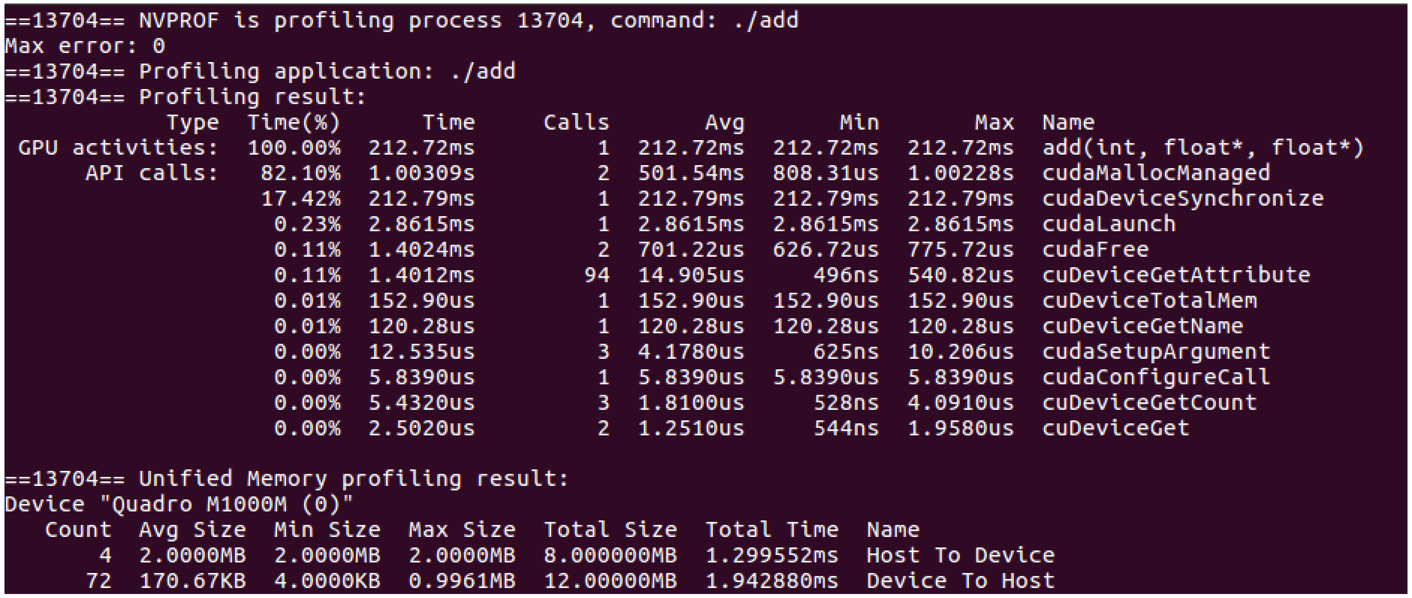
What Happens on Kepler/Maxwell

- After initializing the data on the CPU, the program launches the add() kernel to add the elements of x to the elements of y.

-
  ```
  add<<<1, 256>>>(N, x, y);
  ```
- Upon launching a kernel:
  - the CUDA runtime must migrate all pages previously migrated to host memory or to another GPU back to the device memory of the device running the kernel.
  - Since these older GPUs can’t page fault, all data must be resident on the GPU just in case the kernel accesses it (even if it won’t). This means there is potentially migration overhead on each kernel launch.

What Happens on Kepler/Maxwell

- Note, however, that:
  - the profiler shows the kernel run time separate from the migration time;
  - since the migrations happen before the kernel runs.



```
==13704== NVPROF is profiling process 13704, command: ./add
Max error: 0
==13704== Profiling application: ./add
==13704== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  212.72ms         1  212.72ms  212.72ms  212.72ms  add(int, float*, float*)
      API calls:   82.10%  1.00309s         2  501.54ms  808.31us  1.00228s  cudaMallocManaged
                   17.42%  212.79ms         1  212.79ms  212.79ms  212.79ms  cudaDeviceSynchronize
                    0.23%  2.8615ms         1  2.8615ms  2.8615ms  2.8615ms  cudaLaunch
                    0.11%  1.4024ms         2  701.22us  626.72us  775.72us  cudaFree
                    0.11%  1.4012ms        94  14.905us     496ns  540.82us  cuDeviceGetAttribute
                    0.01%  152.90us         1  152.90us  152.90us  152.90us  cuDeviceTotalMem
                    0.01%  120.28us         1  120.28us  120.28us  120.28us  cuDeviceGetName
                    0.00%  12.535us         3  4.1780us     625ns  10.206us  cudaSetupArgument
                    0.00%  5.8390us         1  5.8390us  5.8390us  5.8390us  cudaConfigureCall
                    0.00%  5.4320us         3  1.8100us     528ns  4.0910us  cuDeviceGetCount
                    0.00%  2.5020us         2  1.2510us     544ns  1.9580us  cuDeviceGet

==13704== Unified Memory profiling result:
Device "Quadro M1000M (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB  1.299552ms  Host To Device
      72  170.67KB  4.0000KB  0.9961MB  12.00000MB  1.942880ms  Device To Host
```

What Happens on Kepler/Maxwell

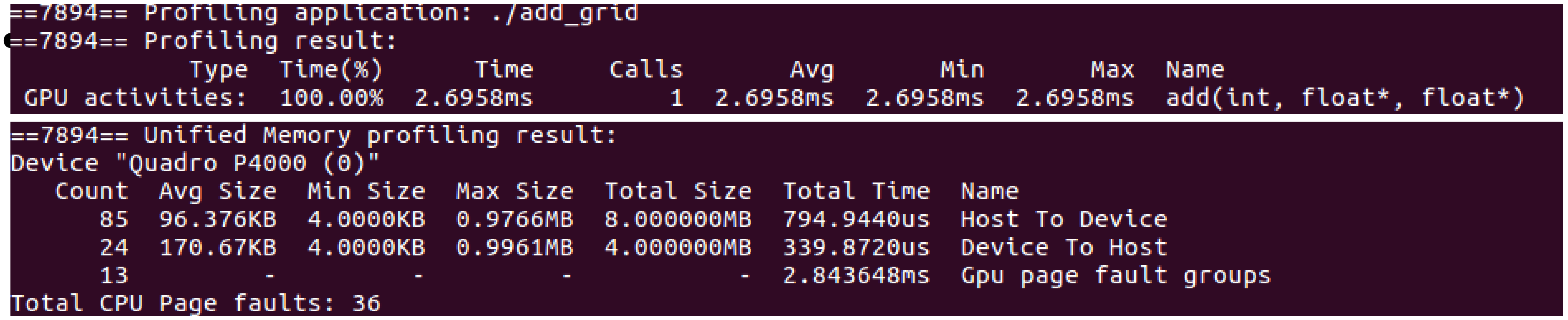
- On Pascal and later GPUs:
  - managed memory may not be physically allocated when cudaMallocManaged() returns; it may only be populated on access.
  - In other words, pages and page table entries may not be created until they are accessed by the GPU or the CPU.
  - The pages can migrate to any processor’s memory at any time, and the driver employs heuristics to maintain data locality and prevent excessive page faults.
-

What Happens on Kepler/Maxwell

- On Pascal and later GPUs:
  - Pascal GPU supports **hardware** page faulting and migration.
  - So in this case the runtime doesn’t automatically copy all the pages back to the GPU before running the kernel.
  - The kernel launches without any migration overhead, and when it accesses any absent pages, the GPU stalls execution of the accessing threads, and the Page Migration Engine migrates the pages to the device before resuming the threads.
-

What Happens on Kepler/Maxwell

- On Pascal and later GPUs:
  - This means that the cost of the migrations is included in the kernel run time when we run program on the P4000 (2.6958 ms).
  - In this kernel, every page in the arrays is written by the CPU, and then accessed by the CUDA kernel on the GPU, causing the kernel to wait on a lot of page migrations.
  - That's why the kernel time measured by the profiler is longer on a Pascal GPU like Quadro P4000.



```
==7894== Profiling application: ./add_grid
==7894== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  2.6958ms         1  2.6958ms  2.6958ms  2.6958ms  add(int, float*, float*)
```

```
==7894== Unified Memory profiling result:
Device "Quadro P4000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      85  96.376KB  4.0000KB  0.9766MB  8.000000MB  794.9440us  Host To Device
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  339.8720us  Device To Host
      13         -         -         -           -  2.843648ms  Gpu page fault groups
Total CPU Page faults: 36
```

What should we do?

- In a real application:
  - the GPU is likely to perform a lot more computation on data (perhaps many times) without the CPU touching it.
- In this simple code:
  - The migration overhead is caused by the fact that the CPU initializes the data and the GPU only uses it once.
- There are a few different ways that I can eliminate or change the migration overhead:
  - Move the data initialization to the GPU in another CUDA kernel.
  - Run the kernel many times and look at the average and minimum run times.
  - Prefetch the data to GPU memory before running the kernel.

What should we do?

- (1) Initialize the Data in a Kernel:
  - If we move initialization from the CPU to the GPU, the add kernel won't page fault.
  - Here's a simple CUDA C++ kernel to initialize the data.

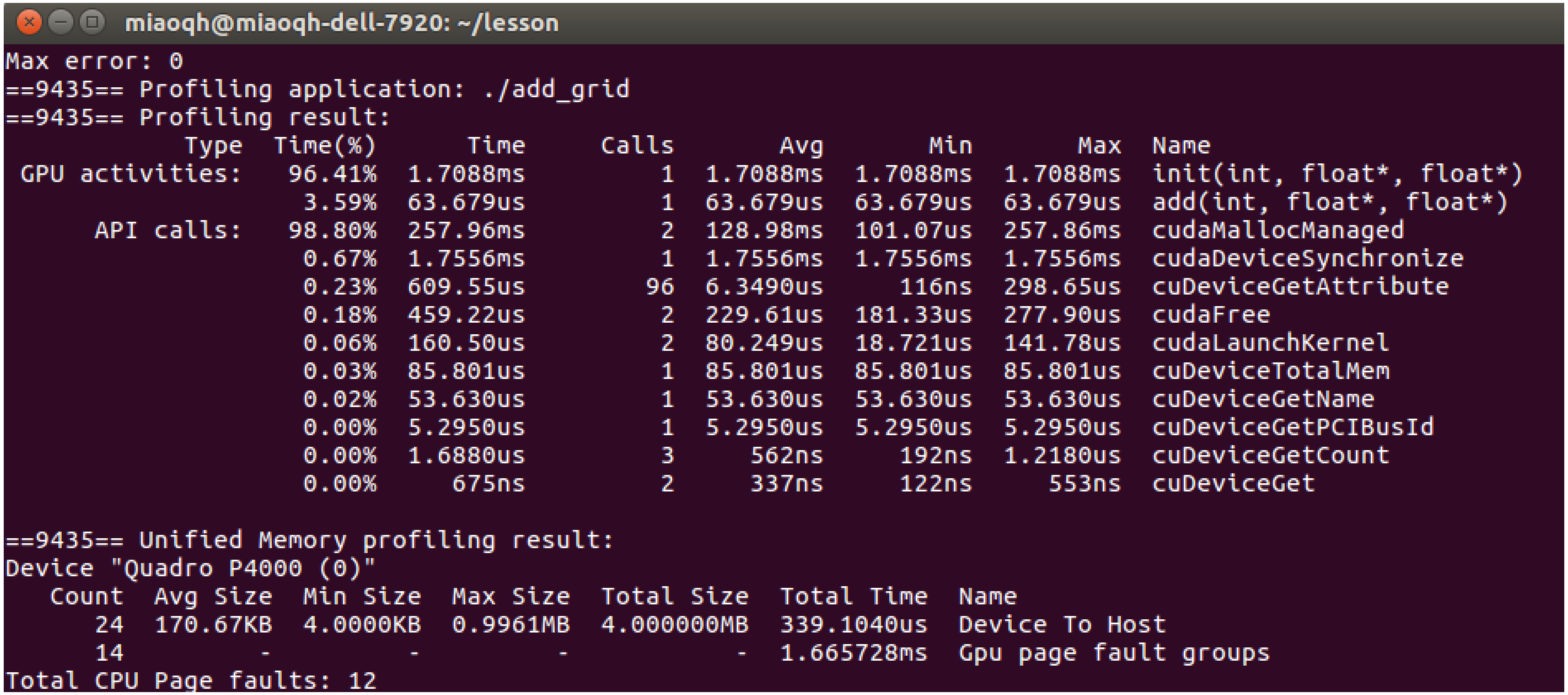
```
__global__
 void init(int n, float *x, float *y) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
}
```

  - We can just replace the host code that initializes x and y with a launch of this kernel.

```
Init<<<1,256>>>(N, x, y);
```

What should we do?

- Both kernels in the profile on the Quadro P4000:



miaoqh@miaoqh-dell-7920: ~/lesson

```
Max error: 0
==9435== Profiling application: ./add_grid
==9435== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   96.41%  1.7088ms         1  1.7088ms  1.7088ms  1.7088ms  init(int, float*, float*)
                    3.59%  63.679us         1  63.679us  63.679us  63.679us  add(int, float*, float*)
      API calls:   98.80%  257.96ms         2  128.98ms  101.07us  257.86ms  cudaMallocManaged
                    0.67%  1.7556ms         1  1.7556ms  1.7556ms  1.7556ms  cudaDeviceSynchronize
                    0.23%  609.55us        96  6.3490us     116ns  298.65us  cuDeviceGetAttribute
                    0.18%  459.22us         2  229.61us  181.33us  277.90us  cudaFree
                    0.06%  160.50us         2  80.249us  18.721us  141.78us  cudaLaunchKernel
                    0.03%  85.801us         1  85.801us  85.801us  85.801us  cuDeviceTotalMem
                    0.02%  53.630us         1  53.630us  53.630us  53.630us  cuDeviceGetName
                    0.00%  5.2950us         1  5.2950us  5.2950us  5.2950us  cuDeviceGetPCIBusId
                    0.00%  1.6880us         3     562ns     192ns  1.2180us  cuDeviceGetCount
                    0.00%     675ns         2     337ns     122ns     553ns  cuDeviceGet

==9435== Unified Memory profiling result:
Device "Quadro P4000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  339.1040us  Device To Host
      14         -         -         -           -  1.665728ms  Gpu page fault groups
Total CPU Page faults: 12
```

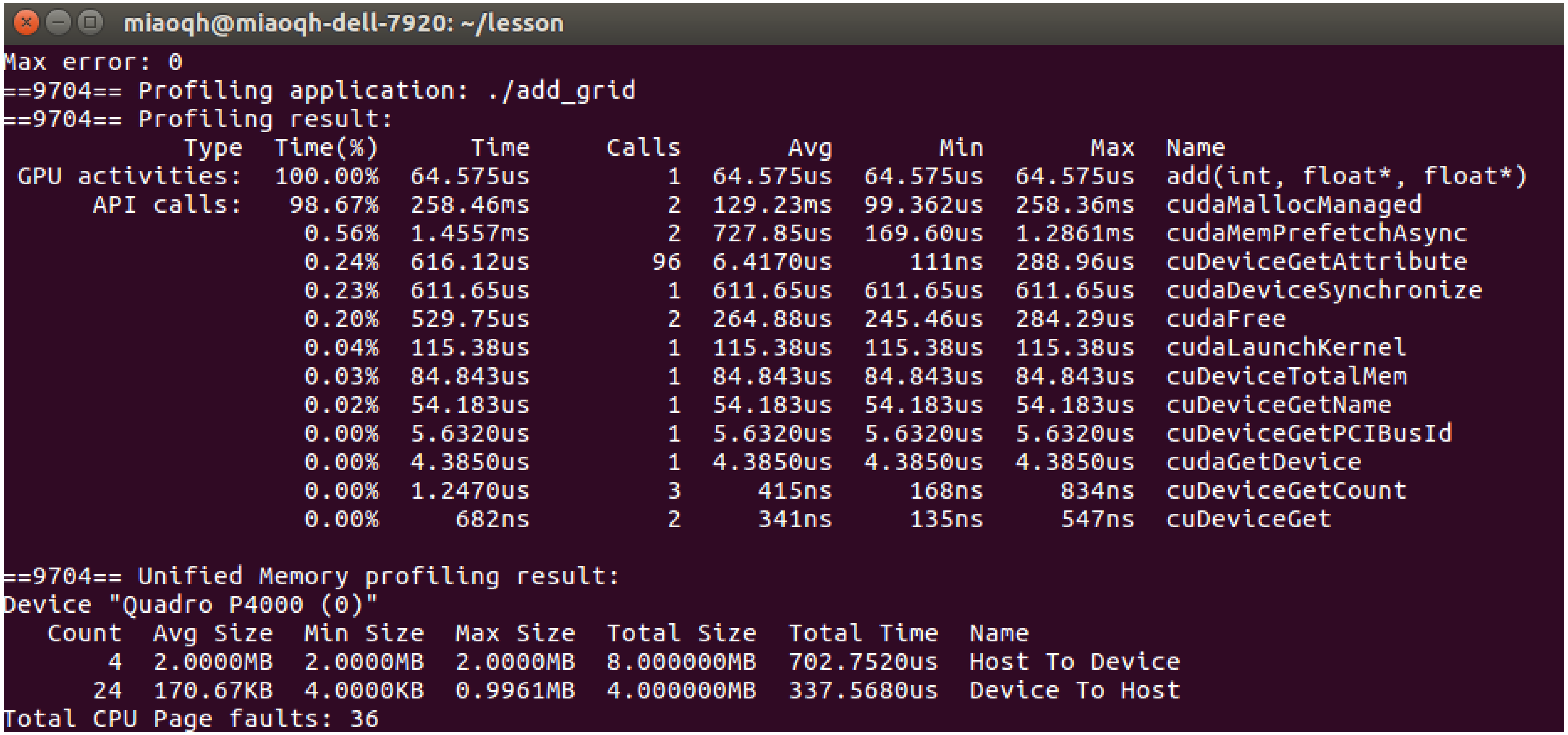
What should we do?

- (2) Prefetching:
  - use Unified Memory prefetching to move data to the GPU after initializing it.
  - CUDA provides cudaMemPrefetchAsync() for this purpose. Add the following code just before the kernel launch.

```
// Prefetch the data to the GPU
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(y, N*sizeof(float), device, NULL);
```

What should we do?

- The profile on the Quadro P4000:



miaoqh@miaoqh-dell-7920: ~/lesson

```
Max error: 0
==9704== Profiling application: ./add_grid
==9704== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  64.575us         1  64.575us  64.575us  64.575us  add(int, float*, float*)
      API calls:   98.67%  258.46ms         2  129.23ms  99.362us  258.36ms  cudaMallocManaged
                    0.56%  1.4557ms         2  727.85us  169.60us  1.2861ms  cudaMemPrefetchAsync
                    0.24%  616.12us        96  6.4170us     111ns  288.96us  cuDeviceGetAttribute
                    0.23%  611.65us         1  611.65us  611.65us  611.65us  cudaDeviceSynchronize
                    0.20%  529.75us         2  264.88us  245.46us  284.29us  cudaFree
                    0.04%  115.38us         1  115.38us  115.38us  115.38us  cudaLaunchKernel
                    0.03%  84.843us         1  84.843us  84.843us  84.843us  cuDeviceTotalMem
                    0.02%  54.183us         1  54.183us  54.183us  54.183us  cuDeviceGetName
                    0.00%  5.6320us         1  5.6320us  5.6320us  5.6320us  cuDeviceGetPCIBusId
                    0.00%  4.3850us         1  4.3850us  4.3850us  4.3850us  cudaGetDevice
                    0.00%  1.2470us         3     415ns     168ns     834ns  cuDeviceGetCount
                    0.00%     682ns         2     341ns     135ns     547ns  cuDeviceGet

==9704== Unified Memory profiling result:
Device "Quadro P4000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
       4  2.0000MB  2.0000MB  2.0000MB  8.000000MB  702.7520us  Host To Device
      24  170.67KB  4.0000KB  0.9961MB  4.000000MB  337.5680us  Device To Host
Total CPU Page faults: 36
```

A Note on Concurrency

- **Keep in mind** that your system has **multiple processors** running parts of your CUDA application concurrently:
  - One or more CPUs and one or more GPUs.
  - Even in our simple example, there is a CPU thread and one GPU execution context.
  - Therefore, ensure there are no race conditions..
- Simultaneous access to managed memory from the CPU and GPUs of compute capability **lower than 6.0 is not possible**.
  - This is because pre-Pascal GPUs lack hardware page faulting, so coherence can’t be guaranteed.
  - On these GPUs, an access from the CPU while a kernel is running will cause a segmentation fault.

A Note on Concurrency

- On Pascal and later GPUs:
  - The CPU and the GPU can simultaneously access managed memory, since they can both handle page faults;
  - however, it is up to the application developer to ensure there are no race conditions caused by simultaneous accesses.
- In our simple example:
  - we have a call to cudaDeviceSynchronize() after the kernel launch.
  - This ensures that the kernel runs to completion before the CPU tries to read the results from the managed memory pointer.
  - Otherwise, the CPU may read invalid data (on Pascal and later), or get a segmentation fault.

Benefits of Unified Memory

- On Pascal and later GPUs:
  - Unified Memory functionality is significantly improved with 49-bit virtual addressing and on-demand page migration.
  - The Page Migration engine allows GPU threads to fault on non-resident memory accesses so the system can migrate pages on demand from anywhere in the system to the GPU’s memory for efficient processing.
- In other words:
  - Unified Memory transparently enables oversubscribing GPU memory, enabling out-of-core computations for any code that is using Unified Memory for allocations.
  - It “just works” without any modifications to the application, whether running on one GPU or multiple GPUs.

Benefits of Unified Memory

- Pascal and Volta GPUs:
  - support system-wide atomic memory operations.
  - That means you can atomically operate on values anywhere in the system from multiple GPUs.
  - This is useful in writing efficient multi-GPU cooperative algorithms.
- Demand paging
  - can be particularly beneficial to applications that access data with a sparse pattern.
  - In some applications, it's not known ahead of time which specific memory addresses a particular processor will access.
  - Page faulting means that only the pages the kernel accesses need to be migrated.