An Introduction To CUDA Programming

HPC Workshop

June 24, 2010, Cetraro, Italy

Piotr Arłukowicz and Janusz Kowalik

piotao@inf.ug.edu.pl

Institute of Informatics

Faculty of Mathematics, Physics and Informatics

University of Gdańsk

Outline

- Hardware
- CUDA Installation
- Software view
- Small example and programming details
- More complex example and more details
- Summary and discussion

Typical hardware configuration

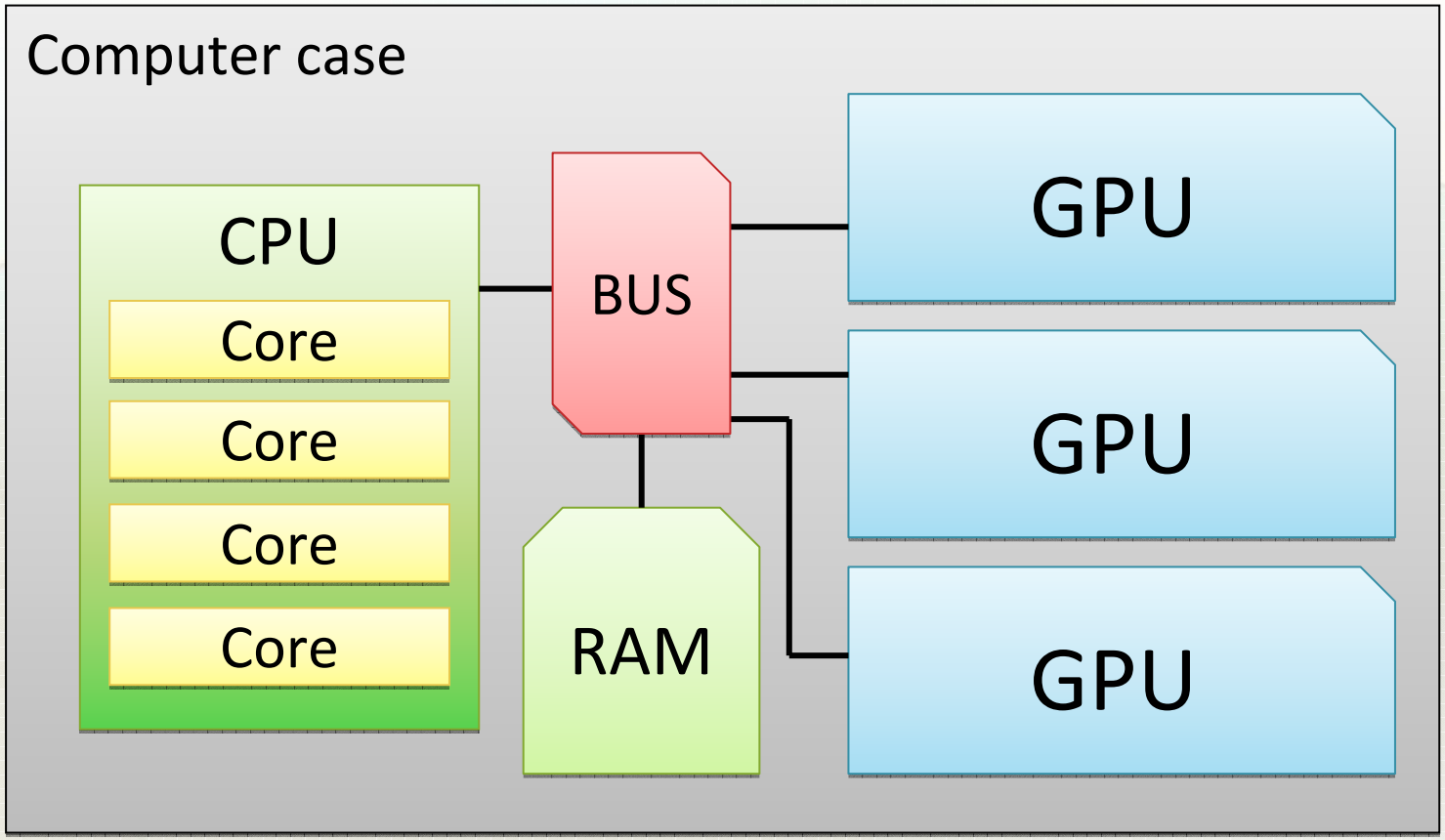
Tesla modular structure

Memory levels and organization

HARDWARE

The typical machine

- Hybrid super-machine can be build at home.



Hardware

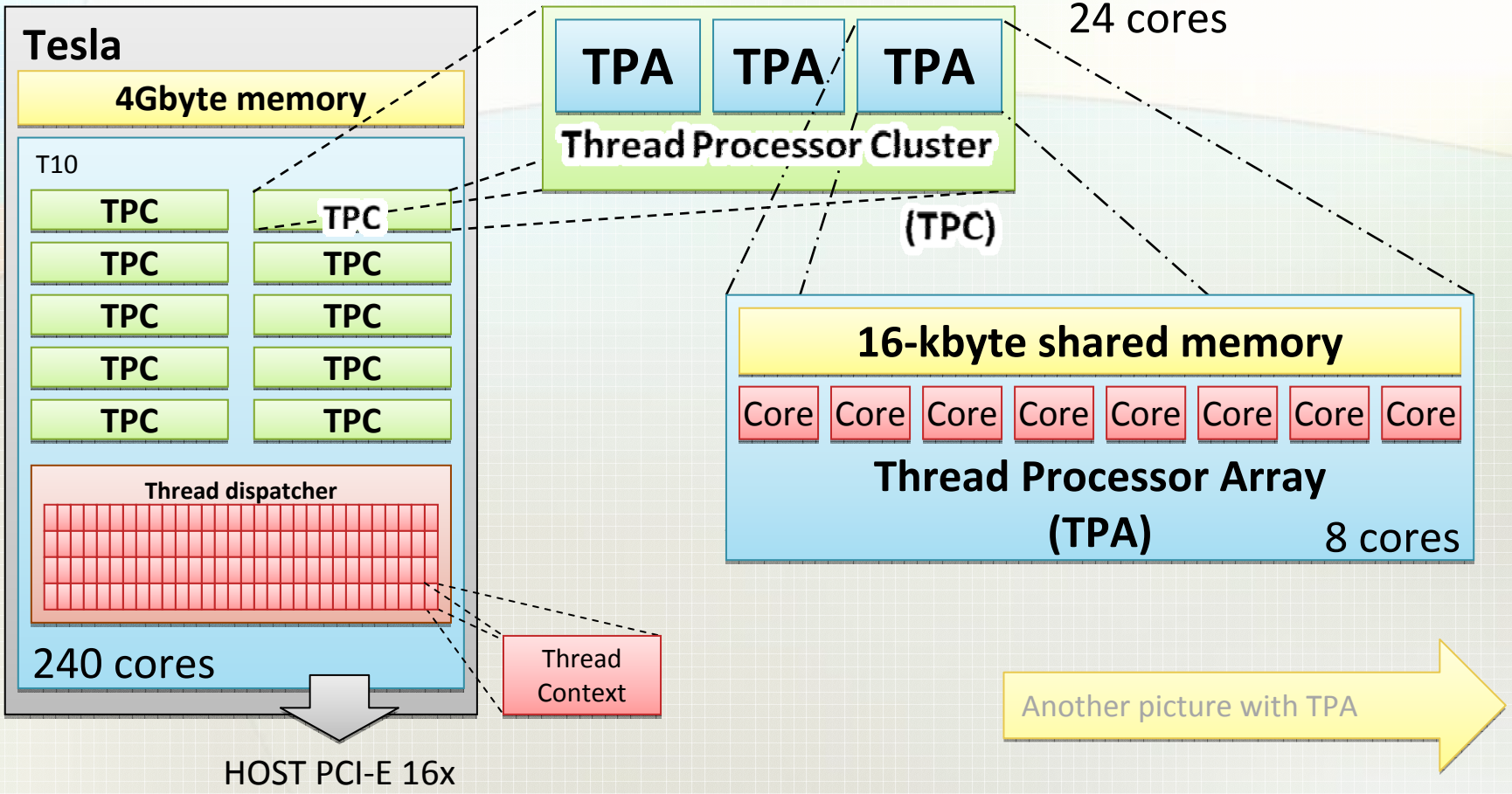
Our test machine

- We have end-user class PC equipped with:
  - Intel CPU i7 920C (4-cores, HT)
  - Tesla C1060 (thanks to courtesy of NVIDIA)
  - NVIDIA GTX 295 (dual GTX285, SLI with Tesla)
  - 12 GB RAM (DDR3, 1066 MHz)
  - Price <2000k$
  - Power Consumed <200W
- Our model has approximate speed:

CPU+Tesla+GTX = 44.8 + 933 + 1788 = **2.8 Tflops**

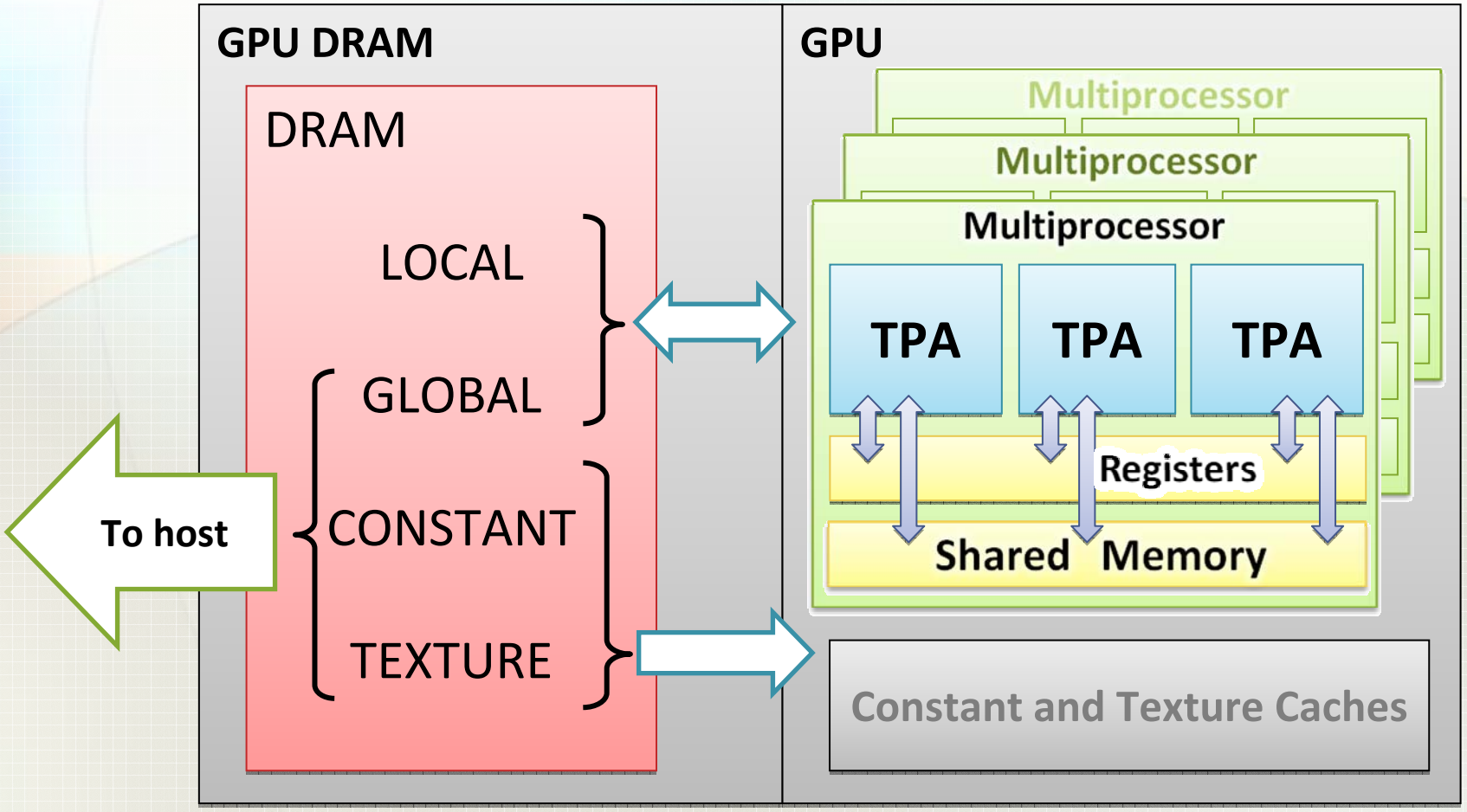
Hardware view

- Let's take a closer look how such a GPU is built.



Hardware view

- Memory organization



How to start

Where to get CUDA

Installing under Windows/Linux

INSTALLATION

Installation

How to start with CUDA

- Buy the device (we got Tesla 1060 for free)
- Plug in PCI-E^{x16} capable motherboard
- Install drivers for your platform
- Install CUDA-SDK/Toolkit
- Write/use the proper software

• Compile! `nvcc`	• Debug! `cuda-gdb`	• Profile! `cuda-prof`

Installation

How to install CUDA

- http://developer.nvidia.com/object/cuda_3_0_downloads.html

[Windows] [Linux] [MacOS]

Windows

Developer Drivers for WinXP (197.13)	32-bit 64-bit	
Developer Drivers for WinVista & Win7 (197.13)	32-bit 64-bit	
Notebook Developer Drivers for WinXP	32-bit 64-bit	
Notebook Developer Drivers for WinVista & Win7	32-bit 64-bit	
CUDA Toolkit • C/C++ compiler • CUDA Visual Profiler • OpenCL Visual Profiler • GPU-accelerated BLAS library • GPU-accelerated FFT library • Additional tools and documentation	32-bit 64-bit	Getting Started Guide for Windows Release Notes CUDA C Programming Guide CUDA C Best Best Practices Guide OpenCL Programming Guide OpenCL Best Best Practices Guide OpenCL Implementation Notes CUDA Reference Manual API Reference PTX ISA 2.0 Visual Profiler User Guide Visual Profiler Release Notes Fermi Compatibility Guide Fermi Tuning Guide CUBLAS User Guide CUFFT User Guide License
NVIDIA Performance Primitives (NPP) library	32-bit 64-bit	
CULA: GPU-accelerated LAPACK libraries	download	more info
NVIDIA Parallel Nsight for Visual Studio		more info
CUDA Fortran from PGI	download	more info
GPU Computing SDK code samples	32-bit 64-bit	Release Notes for CUDA C Release Notes for DirectCompute Release Notes for OpenCL CUDA Occupancy Calculator License
NVIDIA OpenCL Extensions		Compiler_Options D3D9 Sharing D3D10 Sharing D3D11 Sharing Device Attribute Query Pragma Unroll

Linux

Developer Drivers for Linux (195.36.15)	32-bit 64-bit	
CUDA Toolkit • C/C++ compiler • cuda-gdb debugger • CUDA Visual Profiler • OpenCL Visual Profiler • GPU-accelerated BLAS library • GPU-accelerated FFT library • Additional tools and documentation		Getting Started Guide for Linux Release Notes for Linux CUDA C Programming Guide CUDA C Best Best Practices Guide OpenCL Programming Guide OpenCL Best Best Practices Guide OpenCL Implementation Notes CUDA Reference Manual API Reference PTX ISA 2.0 CUDA-GDB User Manual Visual Profiler User Guide Visual Profiler Release Notes Fermi Compatibility Guide Fermi Tuning Guide CUBLAS User Guide CUFFT User Guide License
CUDA Toolkit for Fedora 10	32-bit 64-bit	
CUDA Toolkit for RedHat Enterprise Linux 5.3	32-bit 64-bit	
CUDA Toolkit for Ubuntu Linux 9.04	32-bit 64-bit	
CUDA Toolkit for RedHat Enterprise Linux 4.8	32-bit 64-bit	
CUDA Toolkit for OpenSUSE 11.1	32-bit 64-bit	
CUDA Toolkit for SUSE Linux Enterprise Desktop 11	32-bit 64-bit	
NVIDIA Performance Primitives (NPP) library	32-bit 64-bit	
CULA: GPU-accelerated LAPACK libraries	download	more info
CUDA Fortran from PGI	download	more info
GPU Computing SDK code samples and more	download	Release Notes for CUDA C Release Notes for OpenCL CUDA Occupancy Calculator License
NVIDIA OpenCL extensions		Compiler_Options D3D9 Sharing D3D10 Sharing D3D11 Sharing Device Attribute Query Pragma Unroll

Installation

How to install CUDA

- http://developer.nvidia.com/object/cuda_3_0_downloads.html

Installation

CUDA installation outline

Install under Windowz

- Step 0:
  - Download driver and toolkit
- Step 1:
  - Install NVIDIA Driver
- Step 2:
  - Install CUDA Toolkit

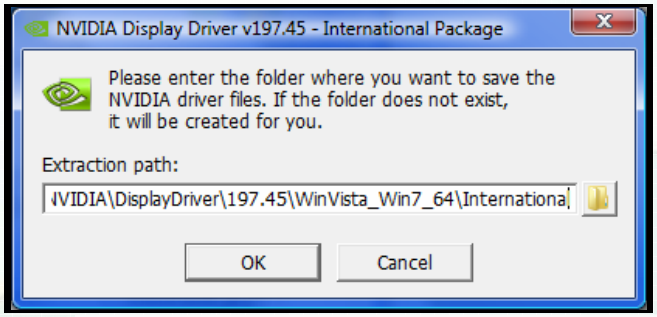
Install under Linux

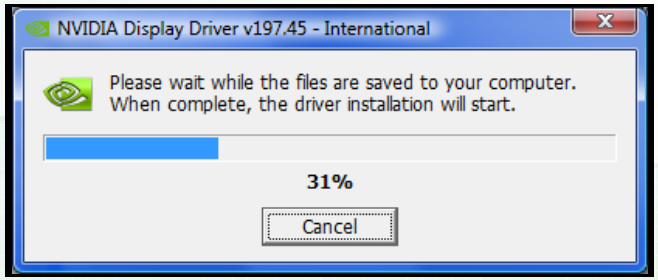
- Step 0:
  - Download „run” file
- Step 1:
  - Run installer

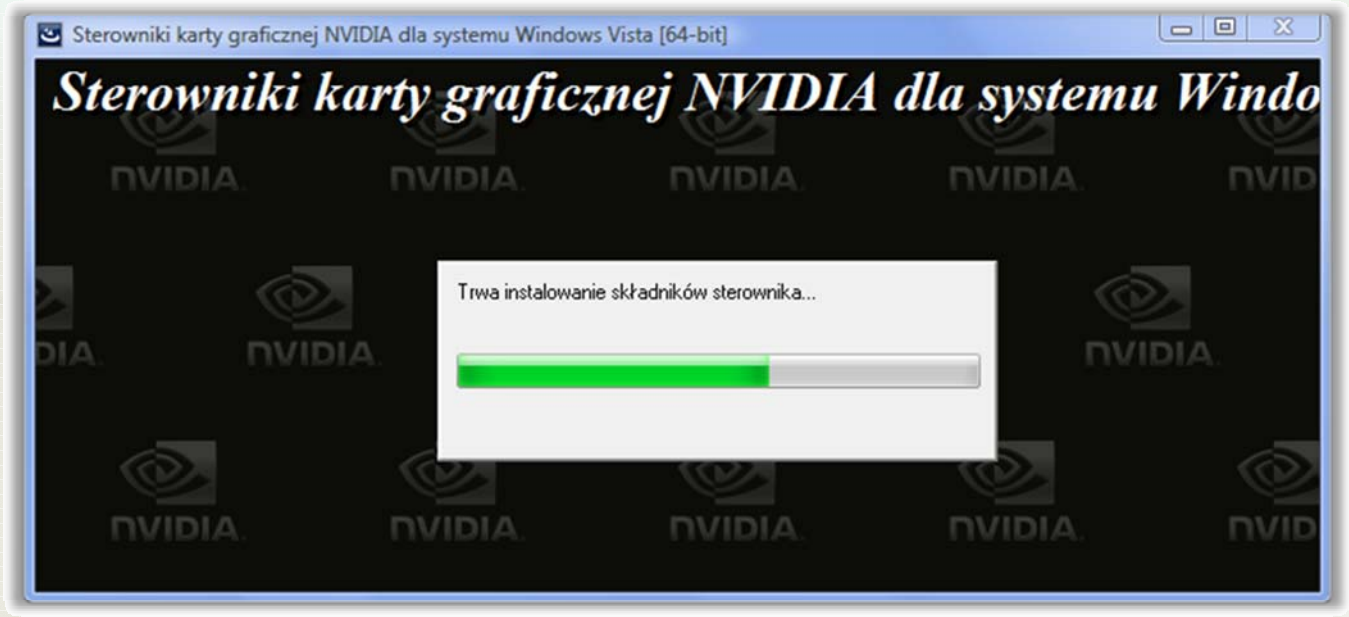
Installation

Installing driver for Windows

- Install drivers for your operating system



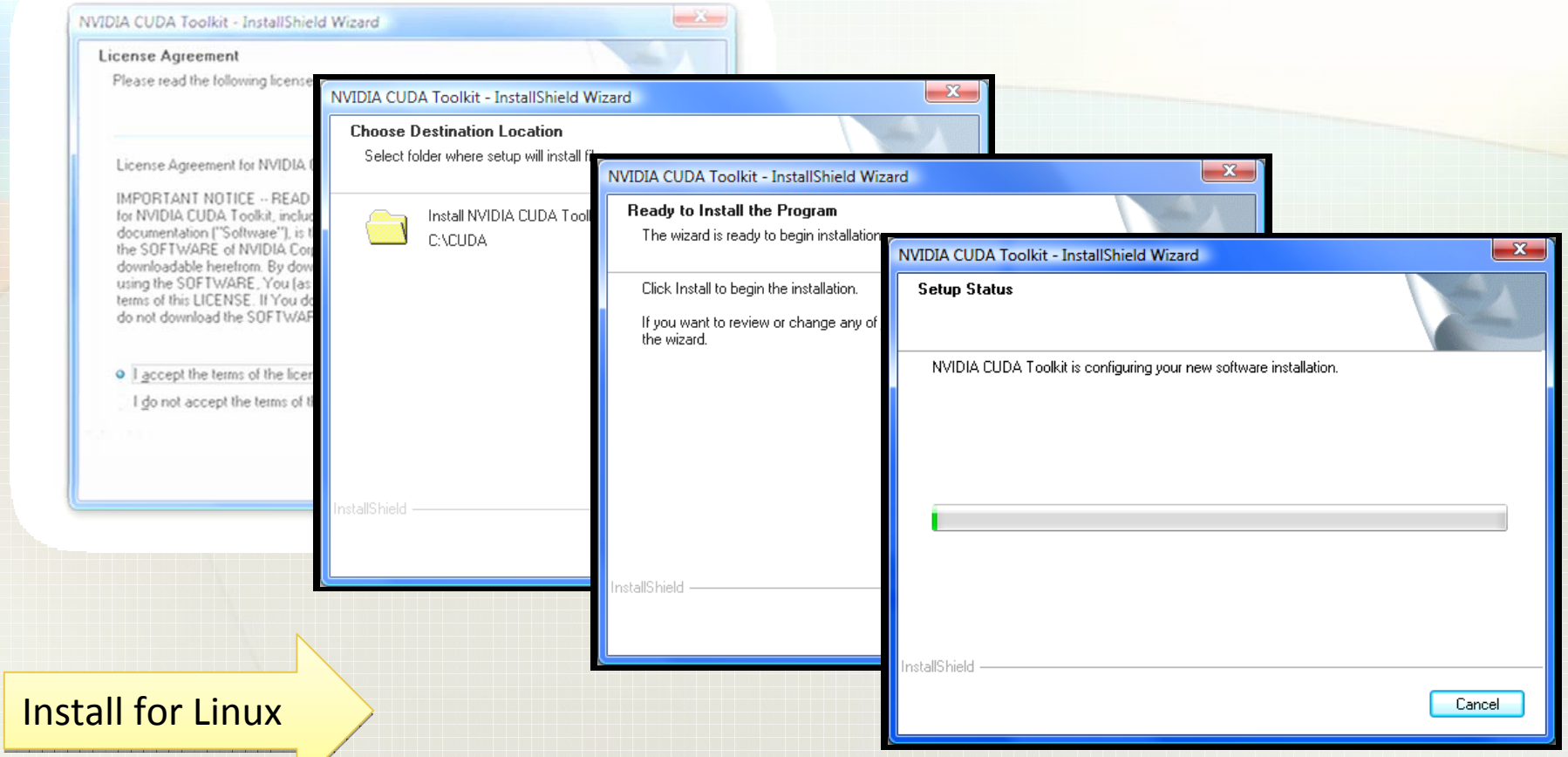




Installation

Installing toolkit for Windows

- Install NVIDIA CUDA Toolkit



Install for Linux

CUDA demo examples

CUDA tools

Compiler Configuration

The „Hello World” program

CUDA SOFTWARE

CUDA Demo Examples

- Demo programs can be found for example in:

```
/opt/cuda
```

- To compile all the demo programs:

```
cd /opt/cuda
make
```

- After compilation, demos can be found in:

```
/opt/cuda/sdk/C/bin/linux/release
```

Software

Compiler and tools

- CUDA compiler driver:
  - nvcc

How nvcc works (diagram)

- CUDA code front end:
  - cudafe
- CUDA profiler:
  - cudaprof
- CUDA assembler:
  - ptxas

PTX example

- CUDA configuration:
  - /opt/cuda/bin/nvcc.profile

Software

nvcc.profile

```
TOP              = $(_HERE_)/..

LD_LIBRARY_PATH += $(TOP)/lib:$(TOP)/extools/lib:/opt/cuda/lib64 …

PATH            += $(TOP)/open64/bin:$(TOP)/bin:

INCLUDES        += "-I$(TOP)/include" \
                   "-I$(TOP)/include/cudart" $(_SPACE_)  \
                   "-I/opt/cuda/sdk/C/common/inc" $(_SPACE_)

LIBRARIES        =+ $(_SPACE_) "-L$(TOP)/lib$(_TARGET_SIZE_)" \
                   -lcudart
CUDAFE_FLAGS    +=
OPENCC_FLAGS    +=
PTXAS_FLAGS     +=
```

How to compile CUDA program

- In order to compile CUDA program:
  - Write your source file as `mysrc.cu`
  - Compile it:

```
nvcc mysrc.cu -o myProgram
```

How nvcc works (diagram)

- You can use many options, i.e.:
  - `-g`
  - `--compiler-options "-Wall -O2"`
- (RTFM)

Hello World, CUDA!

- Prepare a simple test program for CUDA:

```
/* file: hello.cu  */

#include <cuda.h>
int main ( int argc, char **argv ) {
  return 0;
}
```

- And compile it:

```
$ nvcc hello.cu –o hello
```

- And run it:

```
$ ./hello
```

CUDA program structure

Vector addition example

Grids, blocks and threads organization

Matrix multiplication example

Tiling and more details on programming

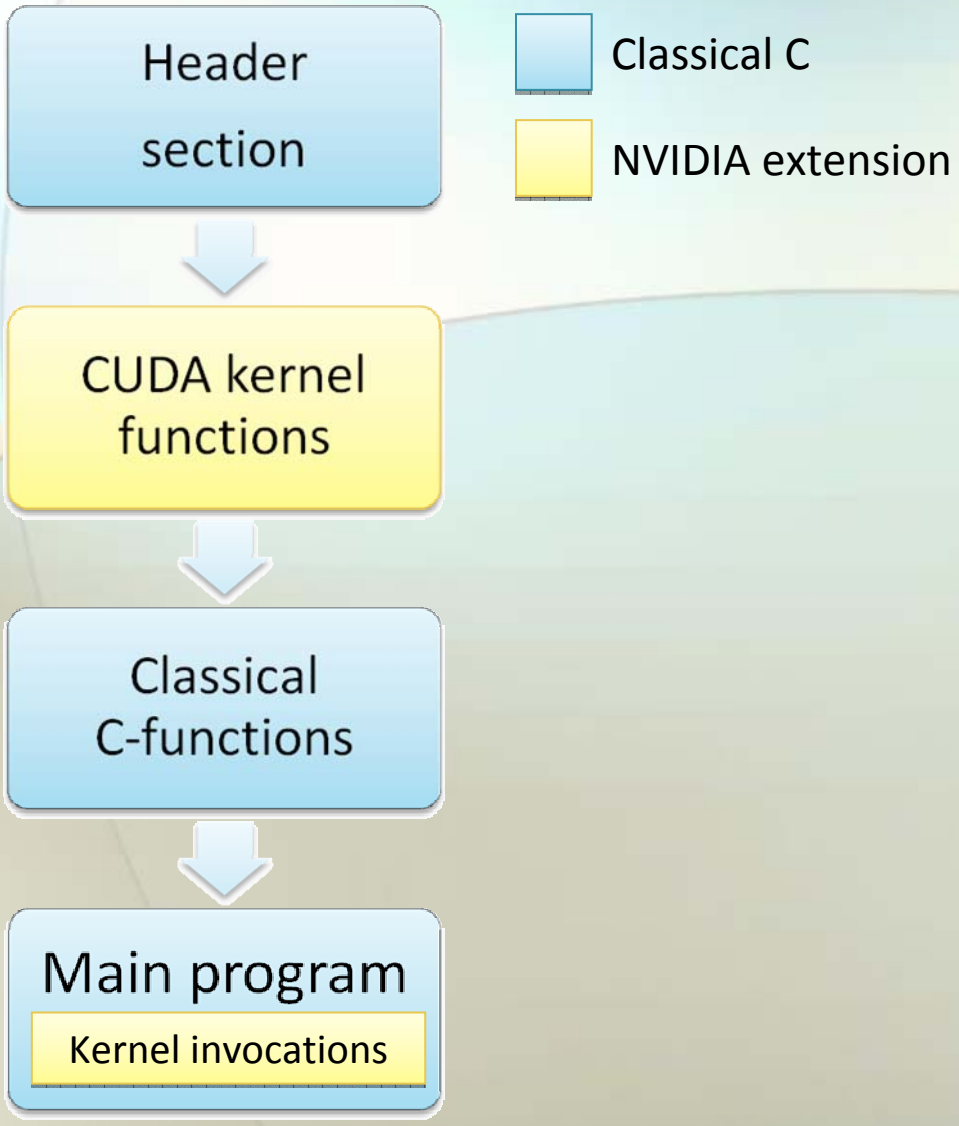
Memory types and shared data

Matrix multiplication optimization

PROGRAMMING

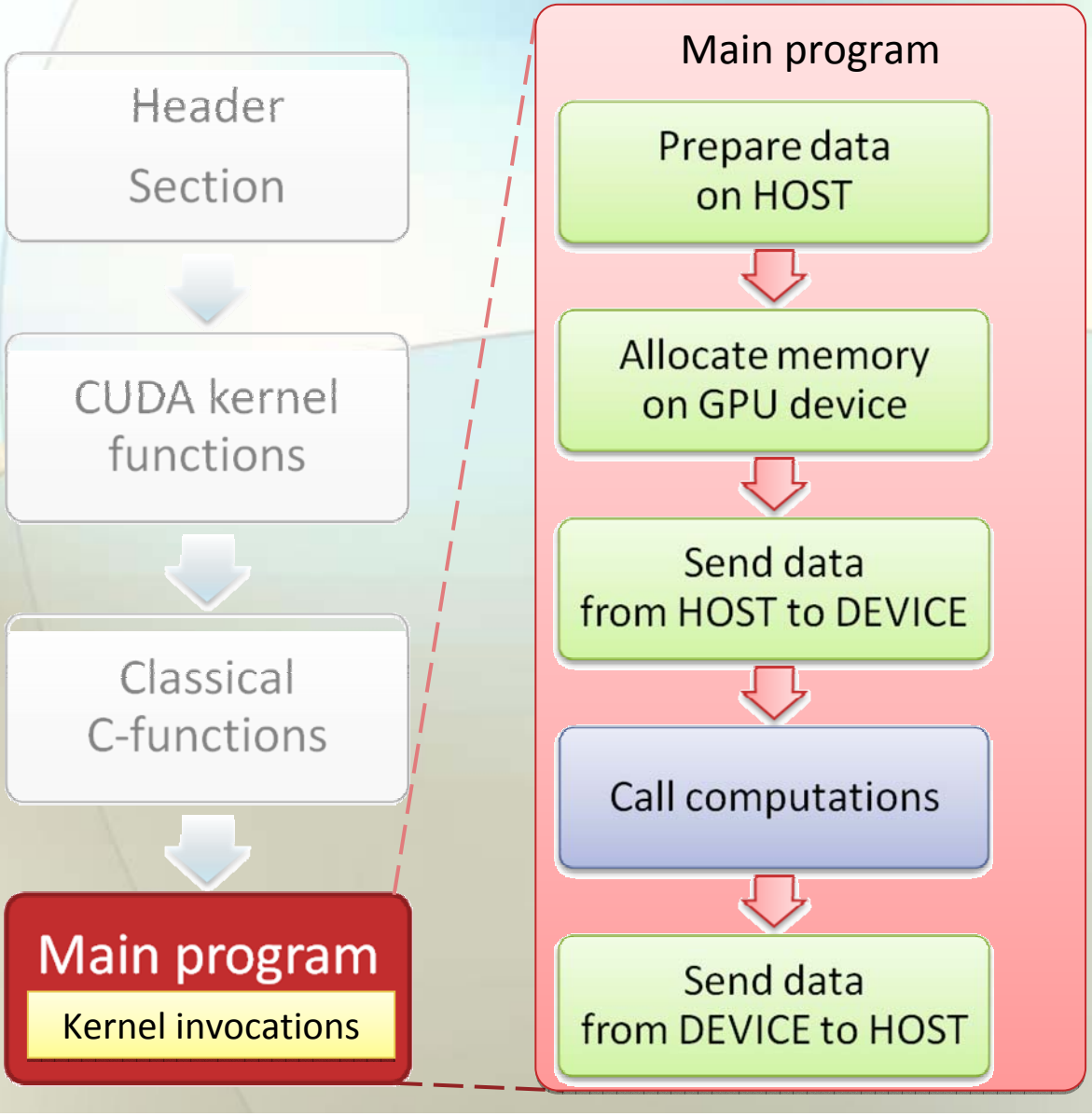
Programming

CUDA program structure



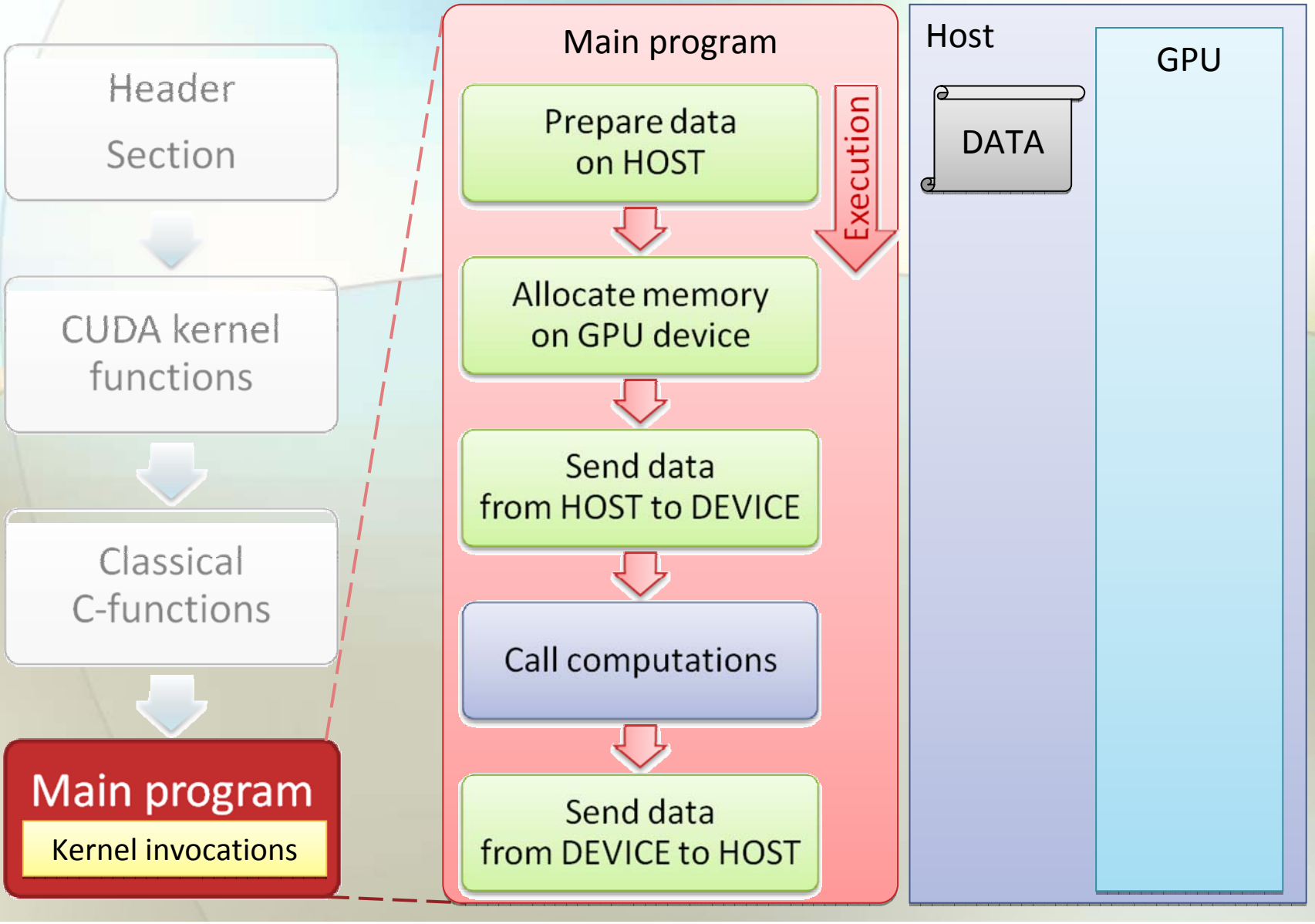
Programming

CUDA program structure



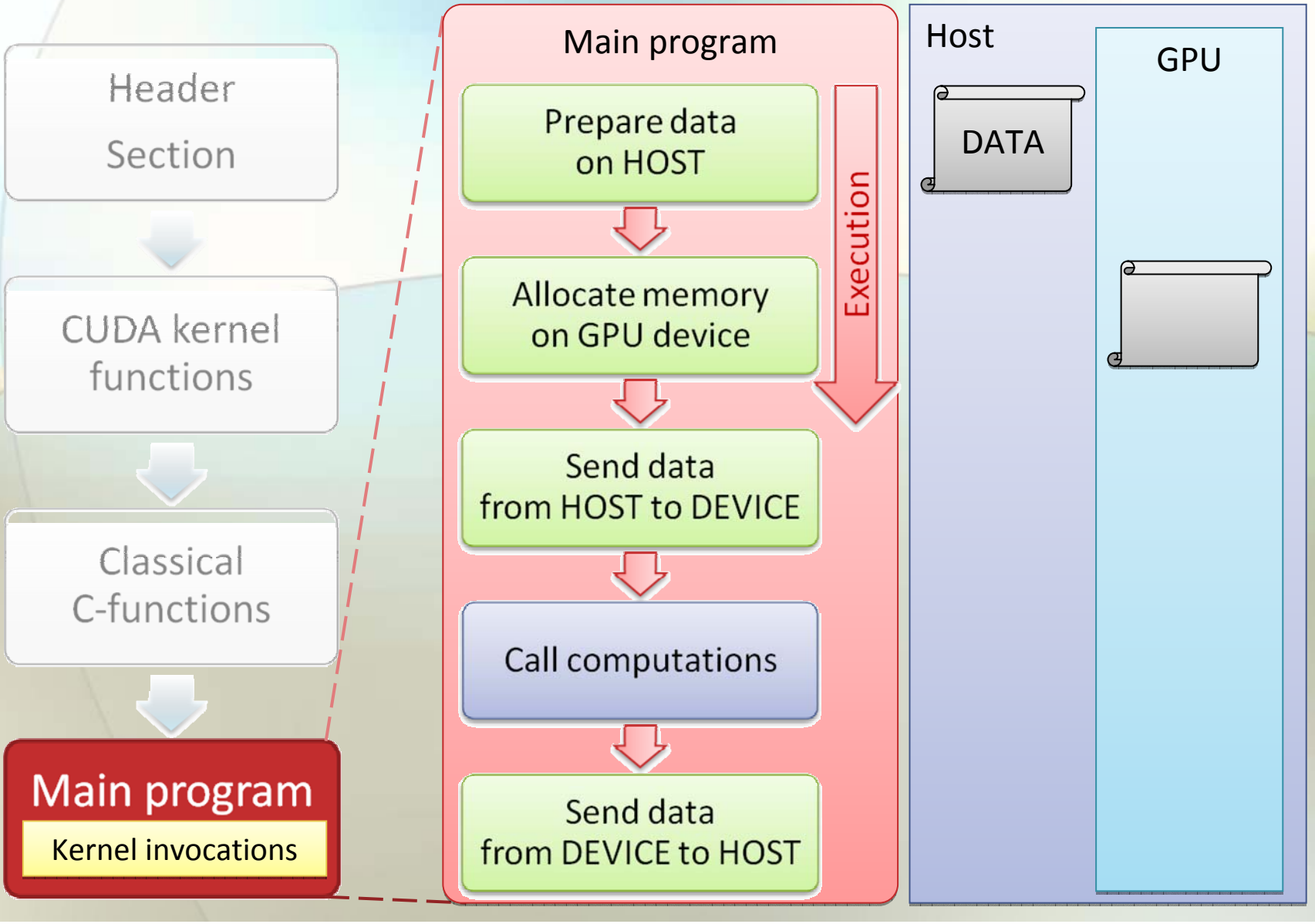
Programming

CUDA program workflow



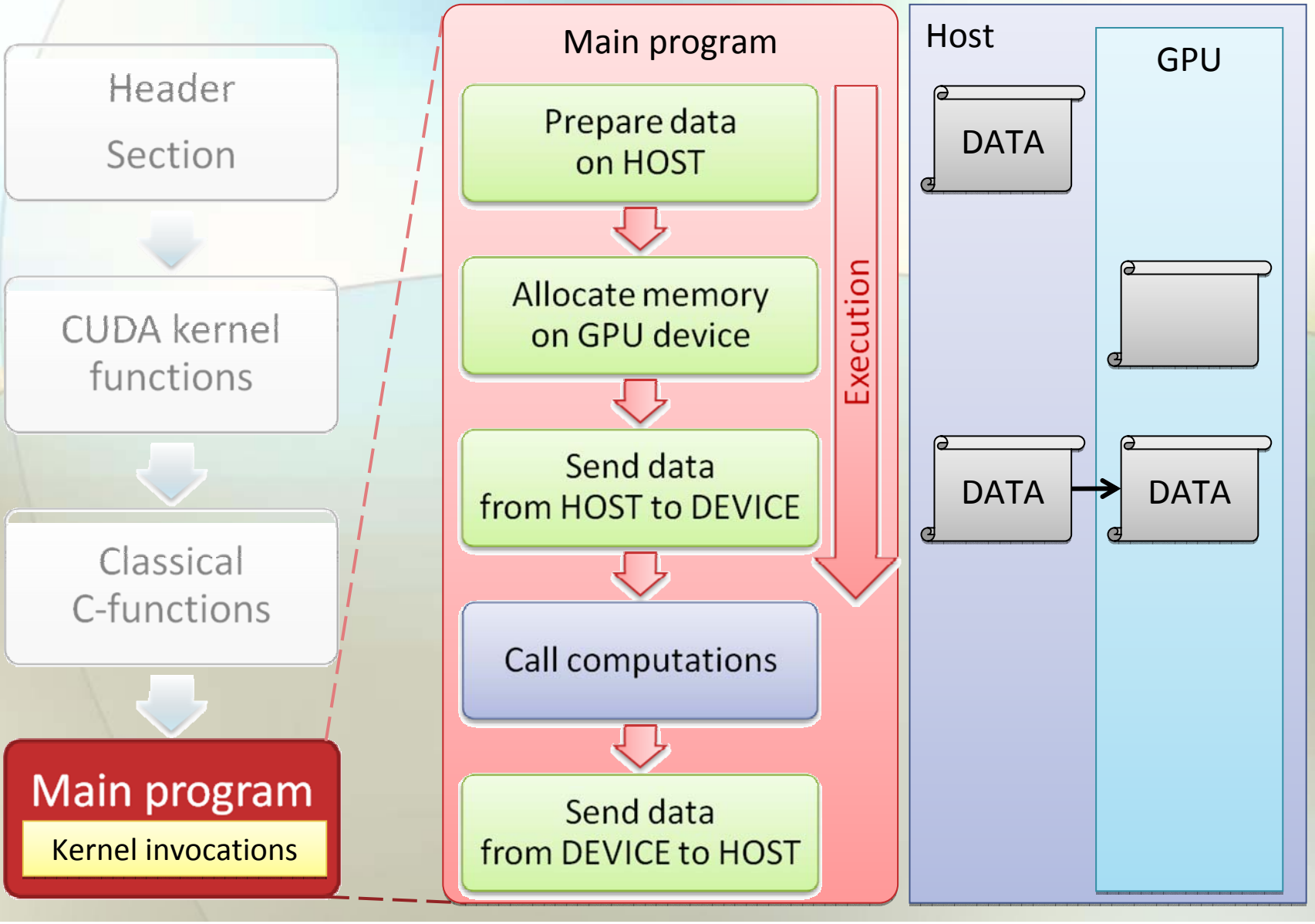
Programming

CUDA program workflow



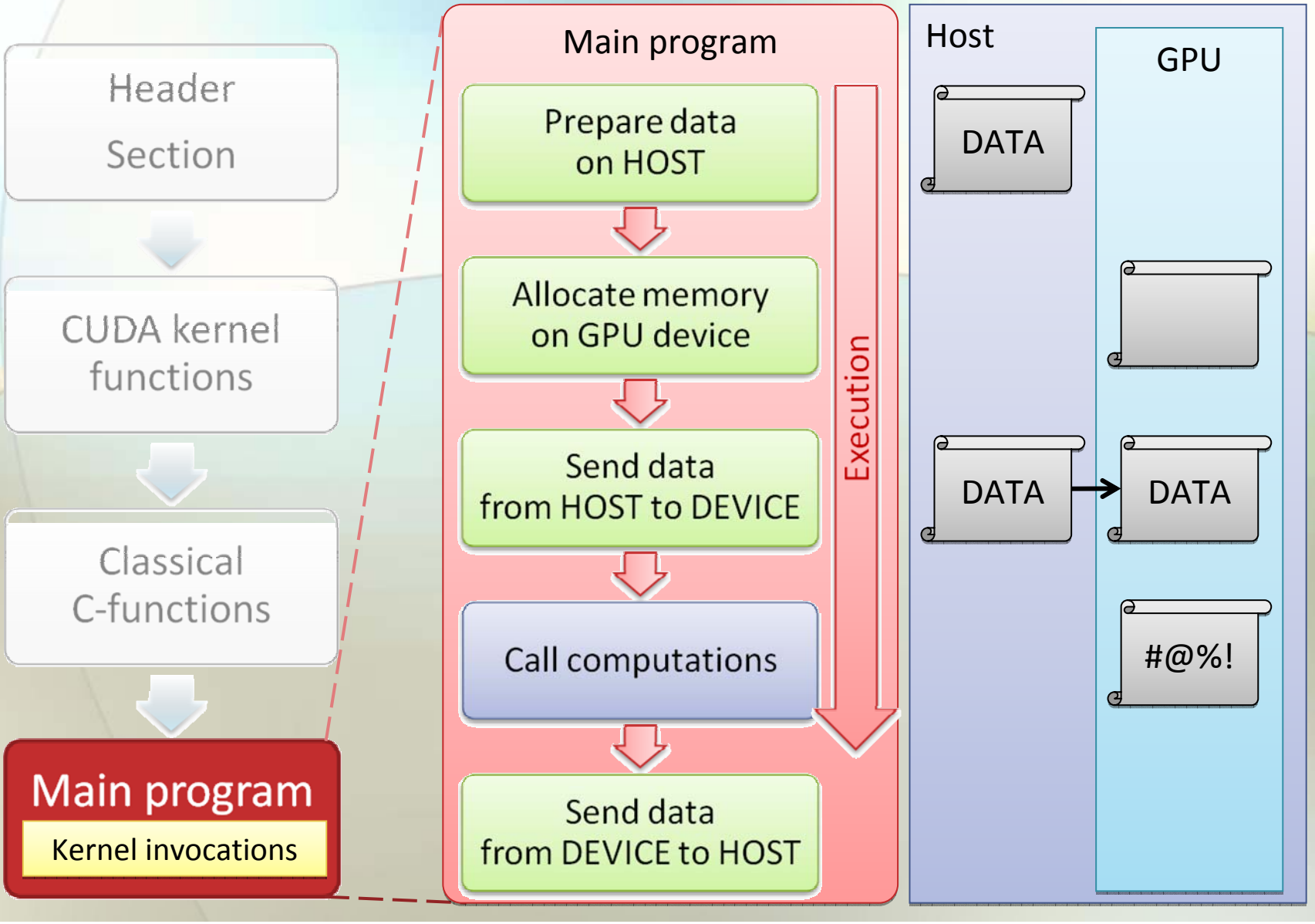
Programming

CUDA program workflow



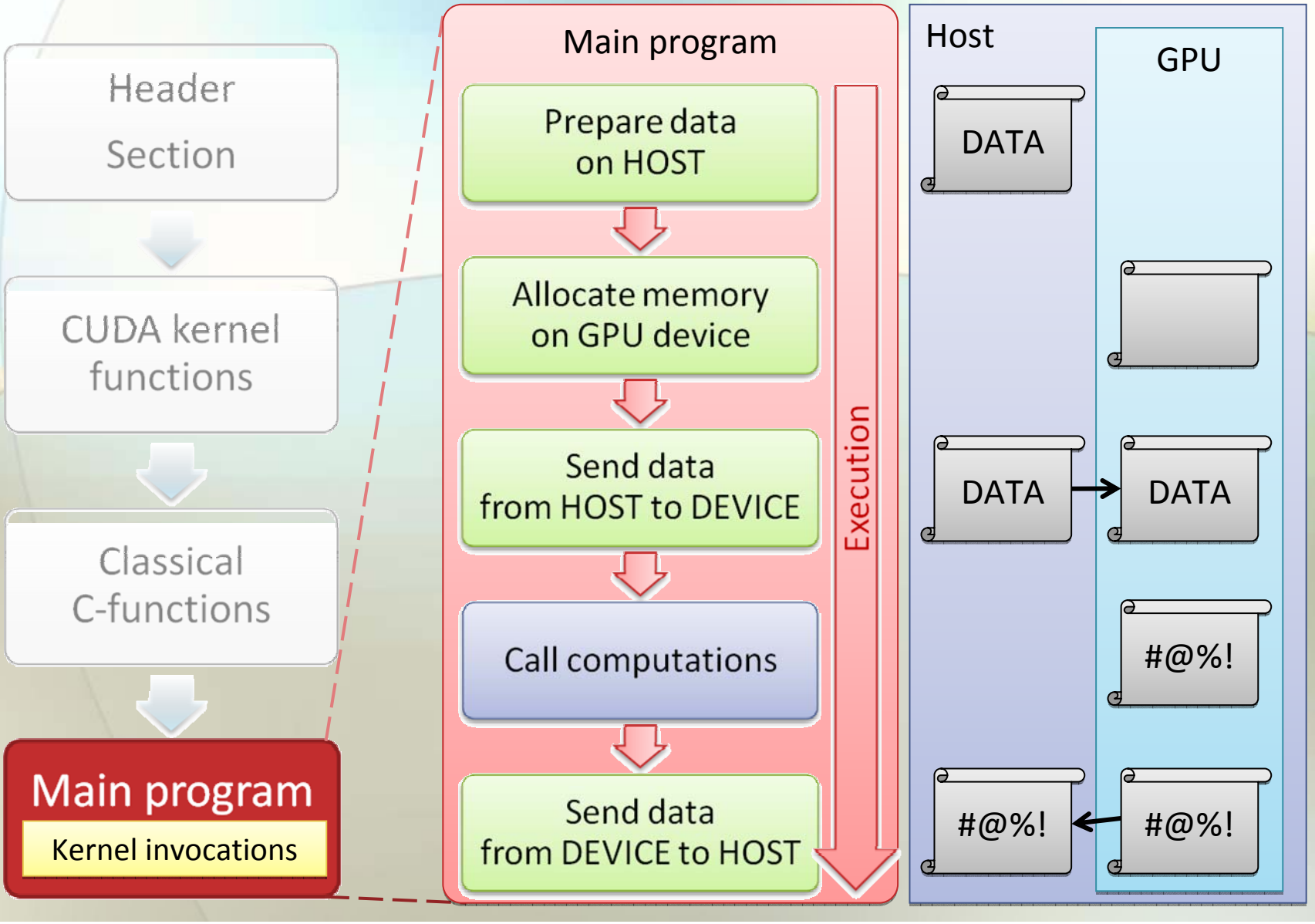
Programming

CUDA program workflow



Programming

CUDA program workflow



CUDA C-program structure

- Each C program written for CUDA has some extensions of standard C language

```
#include ...    /* typical includes */
#include <cuda.h>          /* CUDA includes */


/* CUDA-kernels */
__global__ void kern1 (...) { ... }
__global__ void kern2 (...) { ... }

/* C-functions */
int func (...) { ... }

/* main program section  */
int main ( int argc, char **argv ) {
  a = func(...);                                  /* normal function calls */
  kern2 <<< 32, 16 >>> (...);          /* CUDA kernel calls */
  cudaSyncThreads();                   /* CUDA library function calls */
  return 0;
}
```

Notation vs location

- CUDA introduces few new prefixes to C:

Prefix	Target location
__host__	only host code (default)
__global__	host or device
__device__	device only

- Also:
  - __constant__, __shared__, etc.

Vector addition example

- Lets look how CUDA works using simple vector addition example.
- Assume we have two vectors:
  - hostA = [1, 5, 3, 2, 0, 8, 6, 9, 7, 10];
  - hostB = [9, 4, 5, 5, 6,-3,-2,-6,-5,-1];
- We want to calculate:
  - hostC = hostA + hostB

Vector addition CUDA example

- Solution is trivial in classic C:

```
#define SIZE 10

int main() {
  int hostA[SIZE] = { 1, 5, 3, 2, 0, 8, 6, 9, 7, 10 };
  int hostB[SIZE] = { 9, 4, 5, 5, 6,-3,-2,-6,-5,-1 };
  int hostC[SIZE] = { 0 };
  int i;

  for( i=0; i< SIZE; i++ )
    hostC[i] = hostA[i] + hostB[i];

  return 0;
}
```

One single loop with i as an indexing variable

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {

}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));

}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));

}
```

Vector addition CUDA example

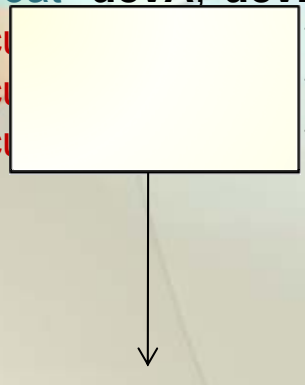
```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data

}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;

}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cu                    **) &devA, SIZE * sizeof(float) );
    cu                    **) &devB, SIZE * sizeof(float) );
    cu                    **) &devC, SIZE * sizeof(float) );

}
```



CUDA library function

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);

}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    V
}
```

CUDA kernel invocation

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    VecAdd<<<1,256>>>(devA,devB,devC);
    cudaMemcpy(hostC,devC,SIZE*sizeof(float),cudaMemcpyDeviceToHost);
}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256



int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    VecAdd<<<1,256>>>(devA,devB,devC);
    cudaMemcpy(hostC,devC,SIZE*sizeof(float),cudaMemcpyDeviceToHost);
    print(hostC);  // now we have data calculated in hostC vector
}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256

// kernel definition
__global__ void VecAdd( float* A, float* B, float* C ){

}

int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    VecAdd<<<1,256>>>(devA,devB,devC);
    cudaMemcpy(hostC,devC,SIZE*sizeof(float),cudaMemcpyDeviceToHost);
    print(hostC);  // now we have data calculated in hostC vector
}
```

Vector addition CUDA example

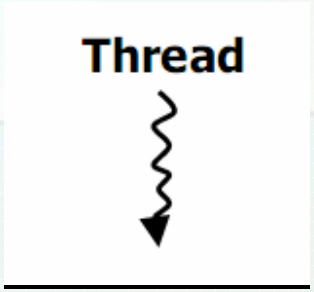
```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256

// kernel definition
__global__ void VecAdd( float* A, float* B, float* C ){

    C[i] = A[i] + B[i];
}

int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    VecAdd<<<1,256>>>(devA,devB,devC);
    cudaMemcpy(hostC,devC,SIZE*sizeof(float),cudaMemcpyDeviceToHost);
    print(hostC);  // now we have data calculated in hostC vector
}
```

Vector addition CUDA example

```
/* source file: vecadd.cu  */
#include <stdio.h>
#include <cuda.h>

#define SIZE 256

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

int main() {
    float *hostA = calloc(SIZE,sizeof(float));
    float *hostB = calloc(SIZE,sizeof(float));
    float *hostC = calloc(SIZE,sizeof(float));
    set(hostA); set(hostB);  // filling vectors with some data
    float *devA,*devB,*devC;
    cudaMalloc((void**) &devA, SIZE * sizeof(float) );
    cudaMalloc((void**) &devB, SIZE * sizeof(float) );
    cudaMalloc((void**) &devC, SIZE * sizeof(float) );
    cudaMemcpy(devA,hostA,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(devB,hostB,SIZE*sizeof(float),cudaMemcpyHostToDevice);
    VecAdd<<<1,256>>>(devA,devB,devC);
    cudaMemcpy(hostC,devC,SIZE*sizeof(float),cudaMemcpyDeviceToHost);
    print(hostC);  // now we have data calculated in hostC vector
}
```

Thread execution

- Now it's time to look under the hood.

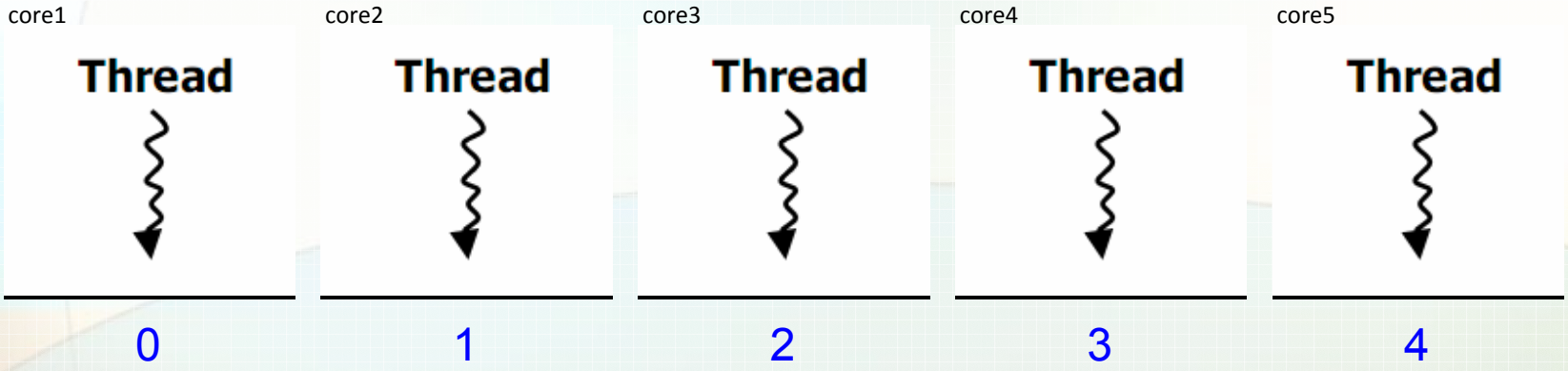



```
__global__ void VecAdd( float* A, float* B, float* C ){
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Thread is executing a simple function: „kernel"

Programming Multiple kernel execution

- Each thread knows its part of data to calculate



← **threadIdx.x** →

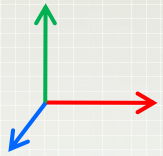
it is just a number of a single thread

```
hostA[  1,      5,      3,      2,      0,  ...];
hostB[  9,      4,      5,      5,      6,  ...];

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

We avoided loop entirely!

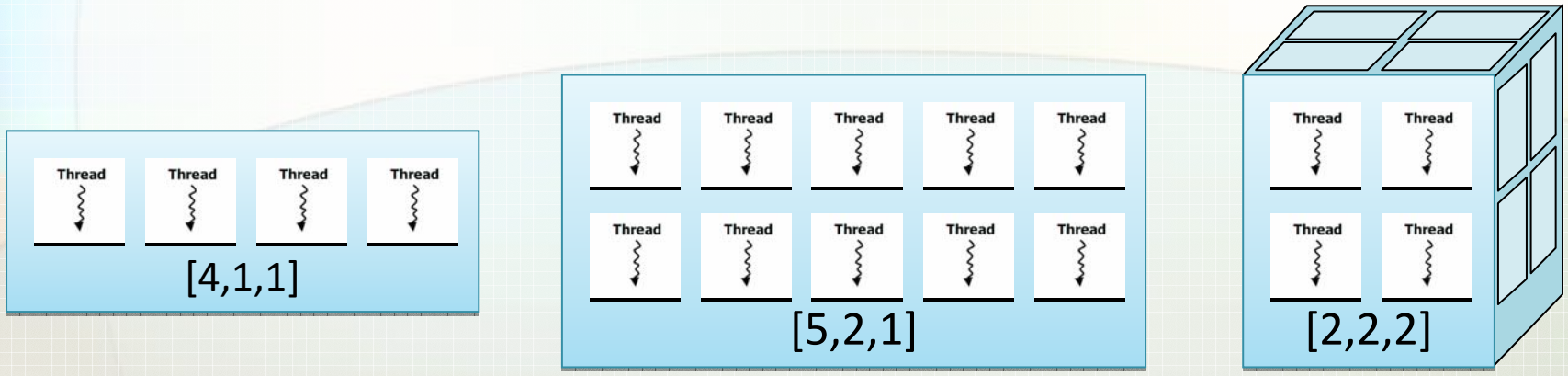
- By the use of many threads we avoided the loop.
- Instead O(n) complexity, we gain O(1)!
- We can execute threads not only in linear fashion.

- Threads can be organized in:
  - one dimension: threadIdx.**x**
  - two dimensions: threadIdx.**x**, threadIdx.**y**
  - three dimensions: threadIdx.**x**, threadIdx.**y**, threadIdx.**z**
- Limits of the run: x=512, y=512, z=64.



Threads can be run in blocks

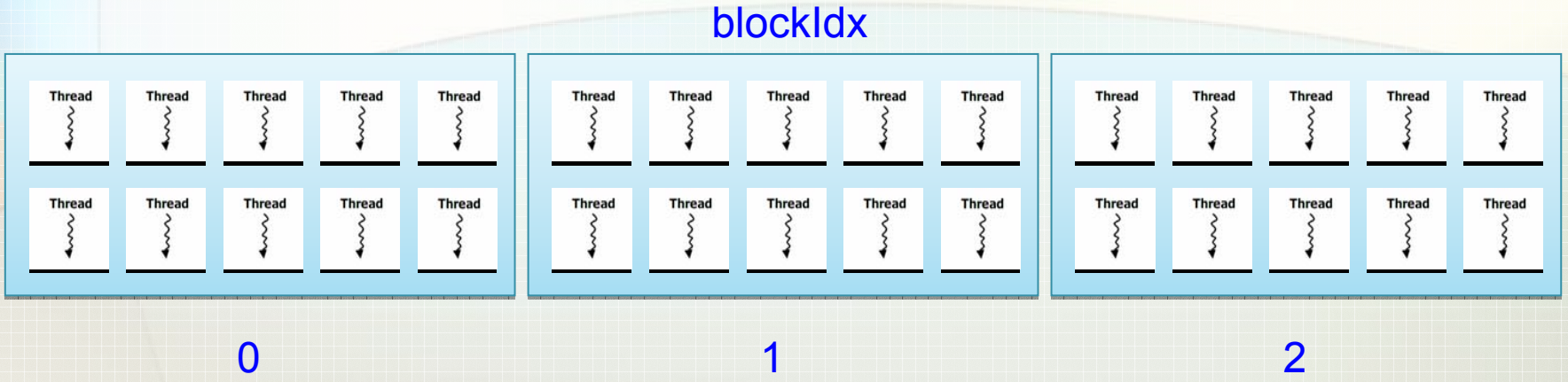
- We can go over the limitation of 512x512x64.
- Threads have to be run in blocks.



- We can then run multiple blocks.
- Therefore blocks also have to have their ID.

Programming
Blocks numbering

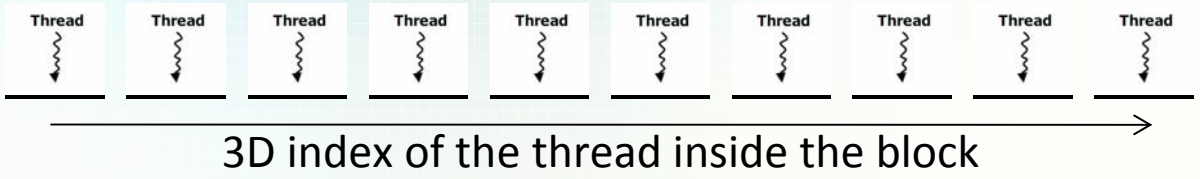
- Threads have their index number in threadIdx.
- Block have their numer in **blockIdx** variable.



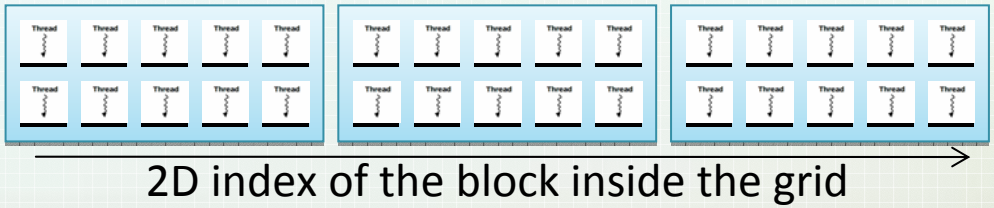
- All threads in a block are executed in parallel.
- All blocks have to have the same dimensions.
- Blocks are executed in grids.

Predefined variables: summary

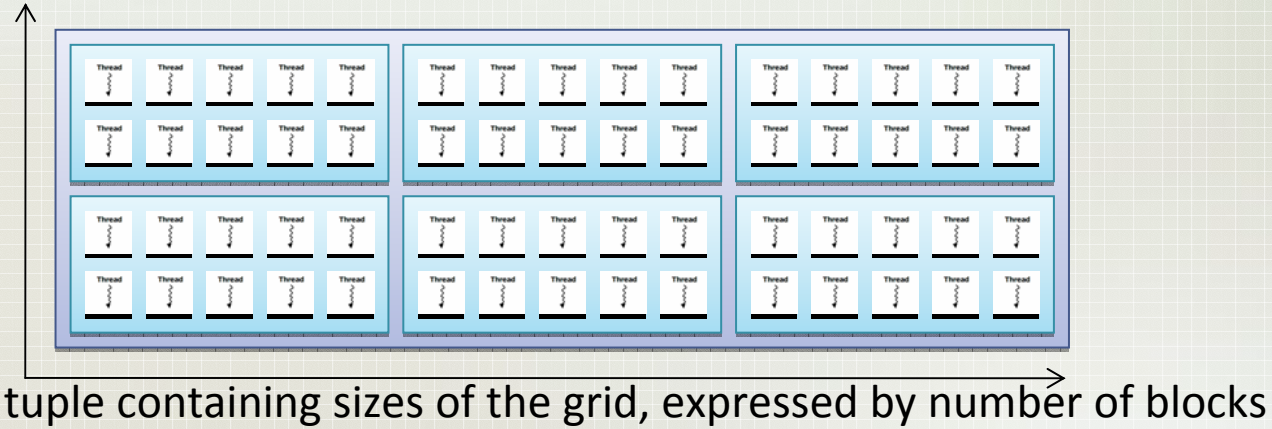
- threadIdx



- blockIdx



- blockDim



Programming Kernel execution

- Now we can execute threads in few different configurations.
- Look at the kernel invocation:

```
kernel <<< [          ], [          ] >>> (. . .);
```

Dimension of the grid in blocks [x,y]

Dimension of the block in threads [x,y,z]

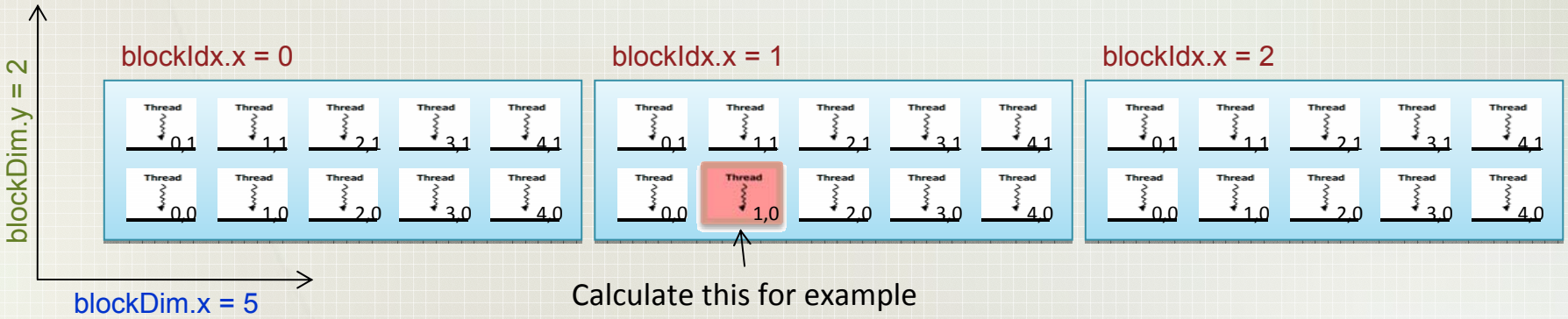
Example:

```
dim3 gridDim(1,3);        dim3 blockDim(2,5);
kernel <<< gridDim, blockDim >>> (…);
```

Programming Indexing model

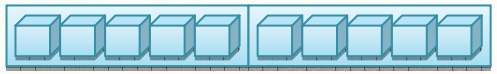
- One dimensional array of blocks on grid where each block has two dimensional array of threads:

```
UniqueThreadIndex =
    blockIdx.x  * blockDim.x * blockDim.y +
    threadIdx.y * blockDim.x + threadIdx.x;
```



Programming
Indexing model, more complicated

- one dimensional array of blocks on grid where each block has three dimensional array of threads then:

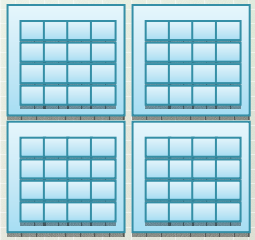


```
UniqueBockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z + threadIdx.z * blockDim.y * blockDim.x +
    threadIdx.y * blockDim.x + threadIdx.x;
```

- two dimensional array of blocks on grid where each block has one dimensional array of threads then:

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;
```

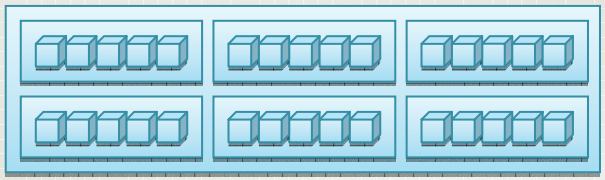
- two dimensional array of blocks on grid where each block has two dimensional array of threads then:



```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y
                * blockDim.x + threadIdx.x;
```

- two dimensional array of blocks on grid where each block has three dimensional array of threads then:

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x + threadIdx.z * blockDim.y *
    blockDim.z + threadIdx.y * blockDim.x + threadIdx.x;
```



Programming
Program results

- Program run for different configurations:

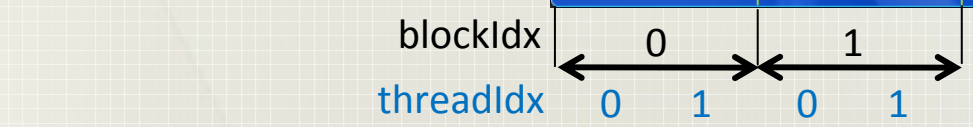
SIZE = 10

gridDim(1,1)
blockDim(3,1)
<<<1,3>>>

```
piotao@maon:~/test
m:~/test> vim main.cu
m:~/test> make
nvcc main.cu -o main
m:~/test> ./main
hostA= 0.19 0.87 0.45 0.04 0.44 0.47 0.40 0.80 0.77 0.65
hostB= 0.77 0.71 0.80 0.32 0.38 0.08 0.58 0.24 0.29 0.41
hostC= 0.96 1.57 1.25 0.00 0.00 0.00 0.00 0.00 0.00 0.00
m:~/test>
```

gridDim(2,1)
blockDim(2,1)
<<<2,2>>>

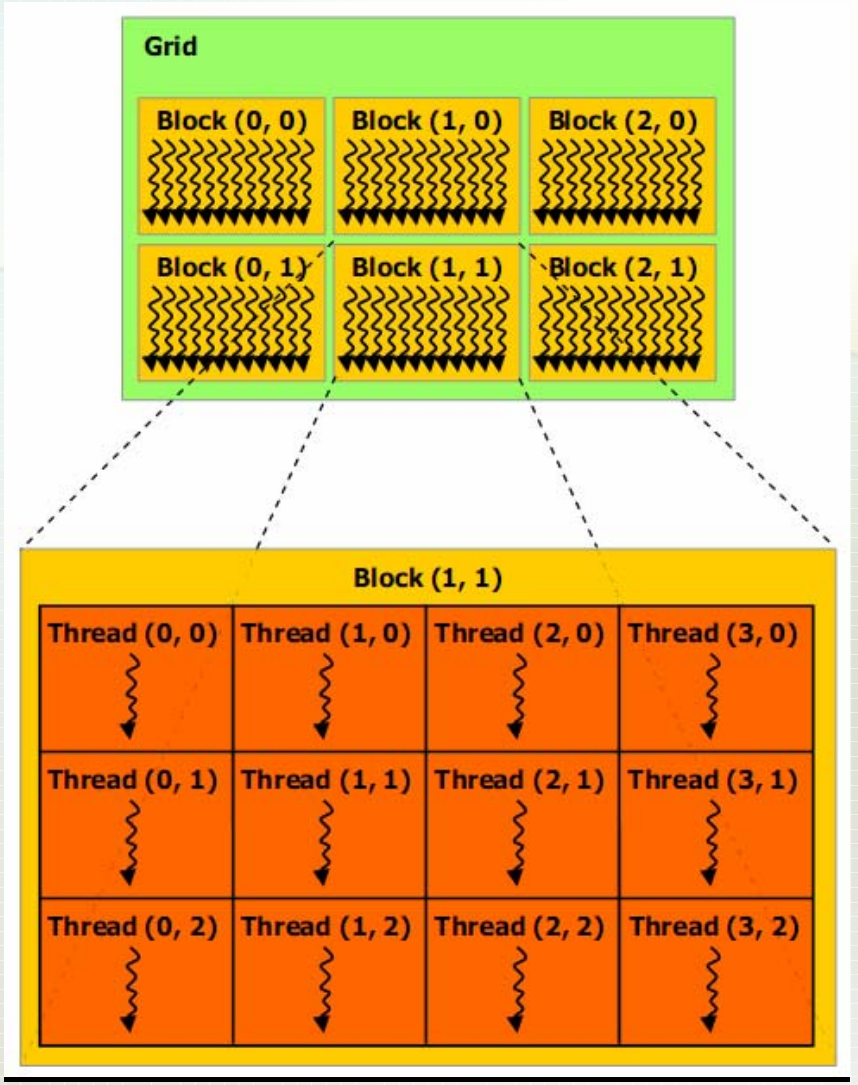
```
piotao@maon:~/test
m:~/test> ./main
hostA= 0.55 0.11 0.65 0.34 0.68 0.06 0.21 0.40 0.19 0.48
hostB= 0.87 0.47 0.63 0.81 0.80 0.46 0.78 0.51 0.69 0.14
hostC= 1.42 0.58 1.28 1.15 0.00 0.00 0.00 0.00 0.00 0.00
m:~/test>
```



Summary of the execution model

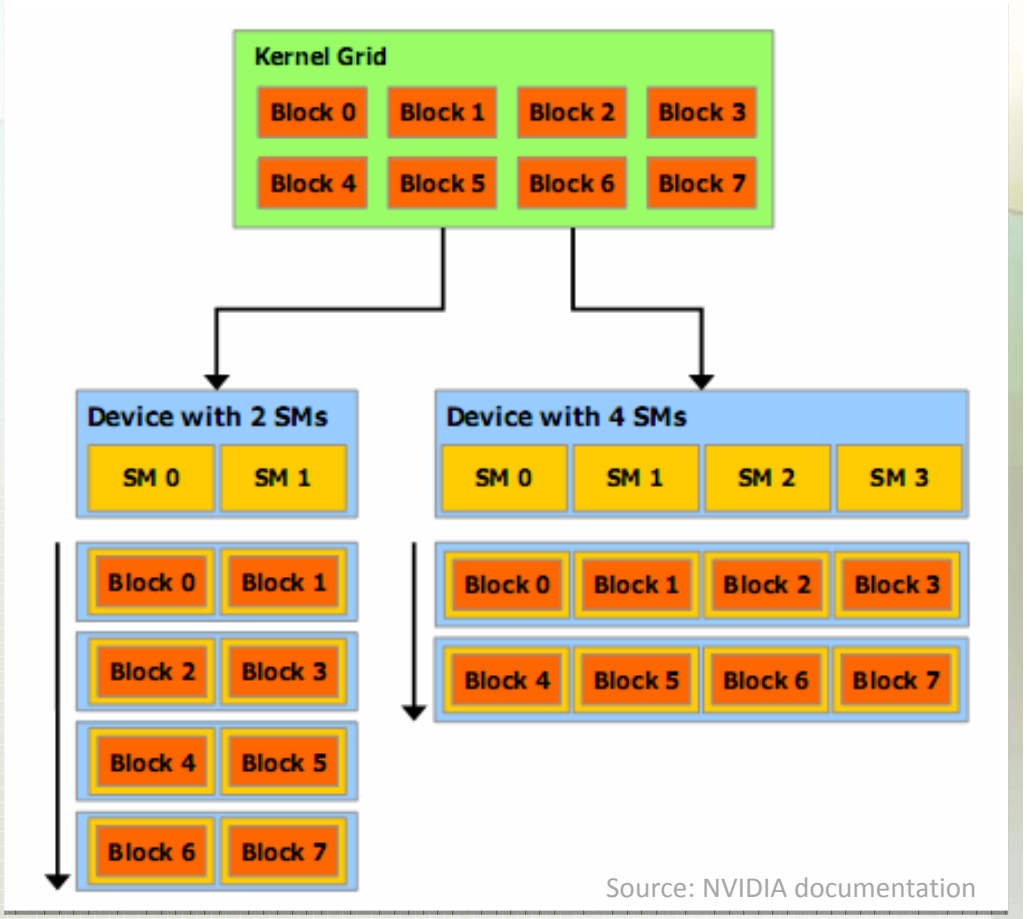
- Tesla C1060:
  - Max threads per block: 512
  - Max blockSize.x = 512
  - Max blockSize.y = 512
  - Max blockSize.z = 64
  - Max gridSize.x = 65535
  - Max gridSize.y = 65535
  - Max registers per block: 16k
  - Max shared mem per block: 16k
  - Total constant memory: 64k
  - Max mem pitch / single copy: 256k
  - Number of cores: 240
  - Number of multiprocessors: 30

Source: NVIDIA documentation



Transparent scalability

- Threads can start in random order
- Blocks are executed in random order
- Multiprocessors provides transparent scalability



Source: NVIDIA documentation

Programming

Matrix multiplication

- Lets look how to multiply matrices.
- We want to calculate the matrix **P**, where:

$$P = N \times M$$

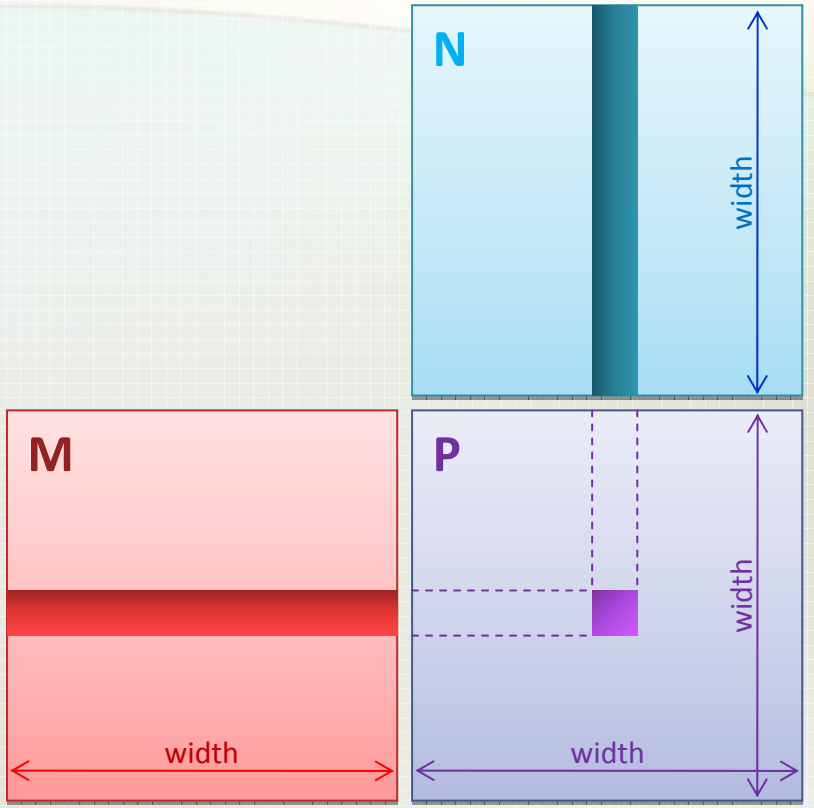
- For simplicity, we are assuming that matrices are square and they have equal dimensions.

Classic matrix multiplication

- Here we have a classical matrix multiplication code:

```
/* simplified notation */

for (i = 0; i < width; i++)
  for (j = 0; j < width; j++)
    for (k = 0; k < width; k++)
      P[i][j] += M[i][k] * N[k][j];
```

```
/* live version from production code */
void mulMatrixRefFast(pMatrix T, pMatrix A, pMatrix B){
  int i, j, k;
  Data sum;
  for (i = 0; i < A->W; i++){
    for (j = 0; j < B->H; j++) {
      sum = (Data) 0.0;
      for (k = 0; k < A->H; k++)
        sum += *(A->T+i*A->H+k) * *(B->T+k*B->H+j);
      *(T->T+i*T->H+j) = sum;
    }
  }
}
```



Programming CUDA matrix multiplication

- First approach: port the code directly into CUDA

```
/* naive CUDA version */
__global__ void naiveMul( float* P, float* M, float* N, int width ) {

  int tx = threadIdx.x;  // current thread position
  int ty = threadIdx.y;  // in 2D block of threads

  float sum = 0.0;       // element computed by the thread

  for (k = 0; k < width; k++)
    sum += M[ty*width+k] * N[k*width+tx];

  P[ty*width+tx] = sum;  // element [tx,ty] is now calculated
}
```

```
...
/* run the kernel */
dim3 blockDim( width, width );
dim3 gridDim(1,1);

naiveMul <<< gridDim, blockDim >>> ( P, M, N, width );    // <<< SIZE, 1 >>>
```

Program discussion

- The code is not using any block index. We are using only **threadIdx** to determine what data in for input and what is for output.
- The entire program is using **width*width** threads:

  dim3 dimBlock(width,width)

- Each grid has only one block:

  dim3 dimGrid(1,1)

- This is OK for small matrices.
  How big could they be in this kernel setup?

Program improvements

- To deal with bigger matrices we have to use more blocks in a grid.
- Each block will calculate a square part of matrix P.
- Each thread in the block will use blockIdx to determine a **tile**, and the threadIdx to determine a data cell to operate on.
- Now we have two-level organization:
  - blockIdx{bx,by} – select a tile
  - threadIdx{tx,ty} – select a thread in a tile

How to tile the matrix

- We have matrix 4x4 matrix.

WIDTH = 4

$P_{[0,0]}$	$P_{[1,0]}$	$P_{[2,0]}$	$P_{[3,0]}$
$P_{[0,1]}$	$P_{[1,1]}$	$P_{[2,1]}$	$P_{[3,1]}$
$P_{[0,2]}$	$P_{[1,2]}$	$P_{[2,2]}$	$P_{[3,2]}$
$P_{[0,3]}$	$P_{[1,3]}$	$P_{[2,3]}$	$P_{[3,3]}$

How to tile the matrix

- We have matrix 4x4 matrix.

$P_{[0,0]}$	$P_{[1,0]}$	$P_{[2,0]}$	$P_{[3,0]}$
$P_{[0,1]}$	$P_{[1,1]}$	$P_{[2,1]}$	$P_{[3,1]}$
$P_{[0,2]}$	$P_{[1,2]}$	$P_{[2,2]}$	$P_{[3,2]}$
$P_{[0,3]}$	$P_{[1,3]}$	$P_{[2,3]}$	$P_{[3,3]}$

WIDTH = 4

TILE_WIDTH = 2

How to tile the matrix

- We have matrix 4x4 matrix.

$P_{[0,0]}$	$P_{[1,0]}$	$P_{[0,0]}$	$P_{[1,0]}$
$P_{[0,1]}$	$P_{[1,1]}$	$P_{[0,1]}$	$P_{[1,1]}$
$P_{[0,0]}$	$P_{[1,0]}$	$P_{[0,0]}$	$P_{[1,0]}$
$P_{[0,1]}$	$P_{[1,1]}$	$P_{[0,1]}$	$P_{[1,1]}$

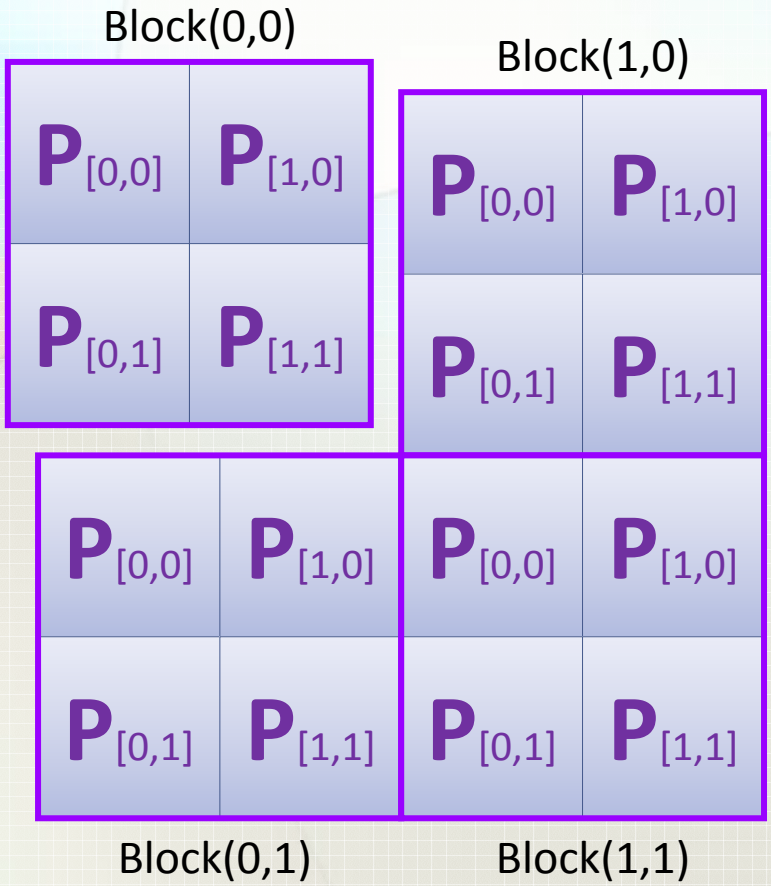
WIDTH = 4

TILE_WIDTH = 2

Note the index change

Tiling by example

- If we have matrix 4x4, tiling it gives 2x2 elements.



WIDTH = 4

TILE_WIDTH = 2

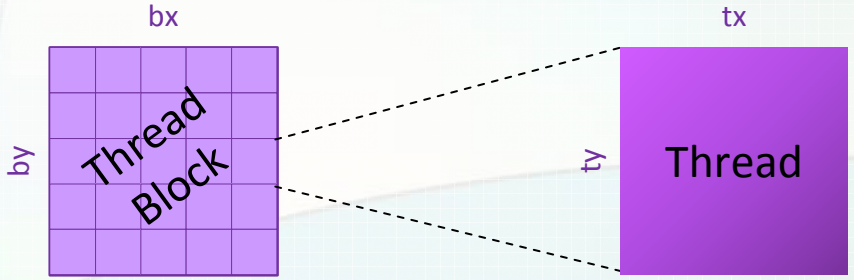
We can access each cell by:

bx * TILE_WIDTH + tx

by * TILE_WIDTH + ty

Programming Matrix tiling

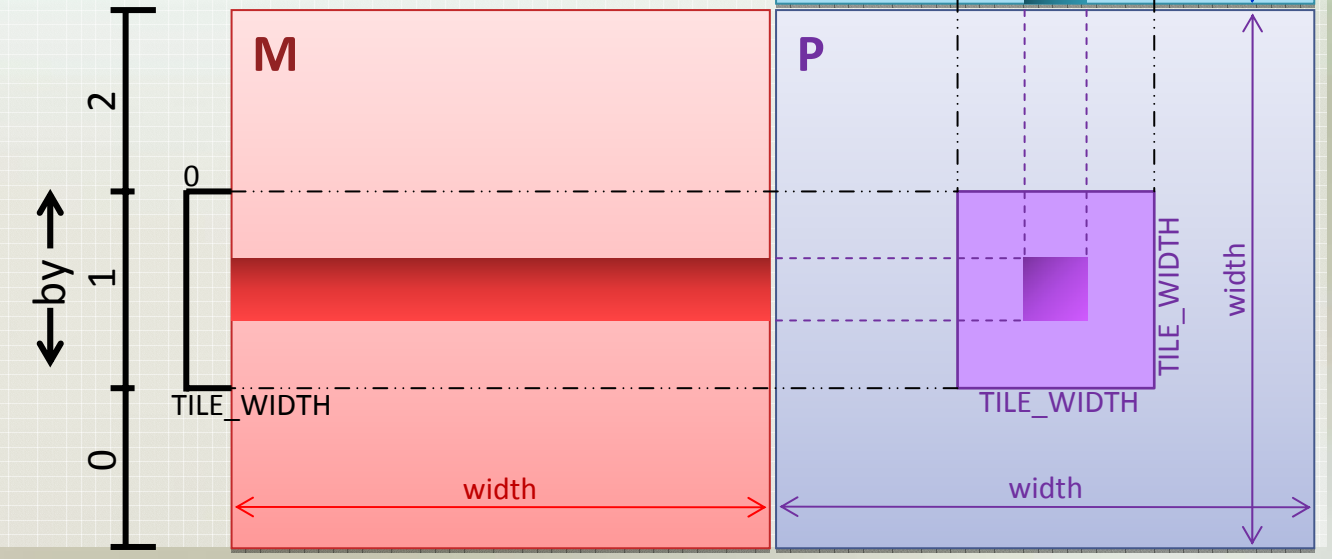
- A tile is handled by a single block.



Block of threads at [bx,by] coordinates in a grid

Single thread at [tx,ty]

```
tx = threadIdx.x;
ty = threadIdx.y;

bx = blockIdx.x;
by = blockIdx.y;
```



Improved kernel function

- We can now improve our kernel to use tiling.

```
/* tiling CUDA kernel*/
__global__ void tilingMul( float* P, float* M, float* N, int width ) {

  int row = blockIdx.y * TILE_WIDTH + threadIdx.y;  // row of P and row of M
  int col = blockIdx.x * TILE_WIDTH + threadIdx.x;  // column of P and N

  float sum = 0.0;       // element computed by the thread

  for (k = 0; k < width; k++)
    sum += M[row*width+k] * N[k*width+col];

  P[row*width+col] = sum;  // element [tx,ty] is now calculated
}

...
/* run the kernel */
dim3 blockDim( width/TILE_WIDTH, width/TILE_WIDTH );
dim3 gridDim(TILE_WIDTH,TILE_WIDTH);

tilingMul <<< gridDim, blockDim >>> ( P, M, N, width );
```

Changes are small: comparision

- Compare changes we introduced.

```
__global__ void naiveMul( P, M, N, width ) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float sum = 0.0;

  for (k = 0; k < width; k++)
    sum += M[ty*width+k] * N[k*width+tx];

  P[ty*width+tx] = sum;

}

...
dim3 blockDim( width, width );
dim3 gridDim(1,1);

naiveMul <<< gridDim, blockDim >>> (P, M, N, width);
```

```
__global__ void tilingMul( P, M, N, width ) {

  int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

  float sum = 0.0;

  for (k = 0; k < width; k++)
    sum += M[row*width+k] * N[k*width+col];

  P[row*width+col] = sum;

}

...
dim3 blockDim( width/TILE_WIDTH, width/TILE_WIDTH );
dim3 gridDim(TILE_WIDTH,TILE_WIDTH);

tilingMul <<< gridDim, blockDim >>> (P, M, N, width);
```

- Now, we can go beyond 16x16 limitation.

Memory access efficiency

- The longest working part of kernel is the loop:

```
for (k = 0; k < width; k++)
  sum += M[row*width+k] * N[k*width+col];
```

- We have here:
  - One floating point multiplication — M * N
  - One floating point addition — sum +=
  - We are ignoring integer calculations here
  - Two global memory accesses — M[…] and N[…]

Memory access efficiency

- So, the ratio between floating point calculations and corresponding memory access is:

$$1 : 1 = 1.0$$

- This ratio is called

Compute to **G**lobal **M**emory **A**ccess Ratio

CGMA

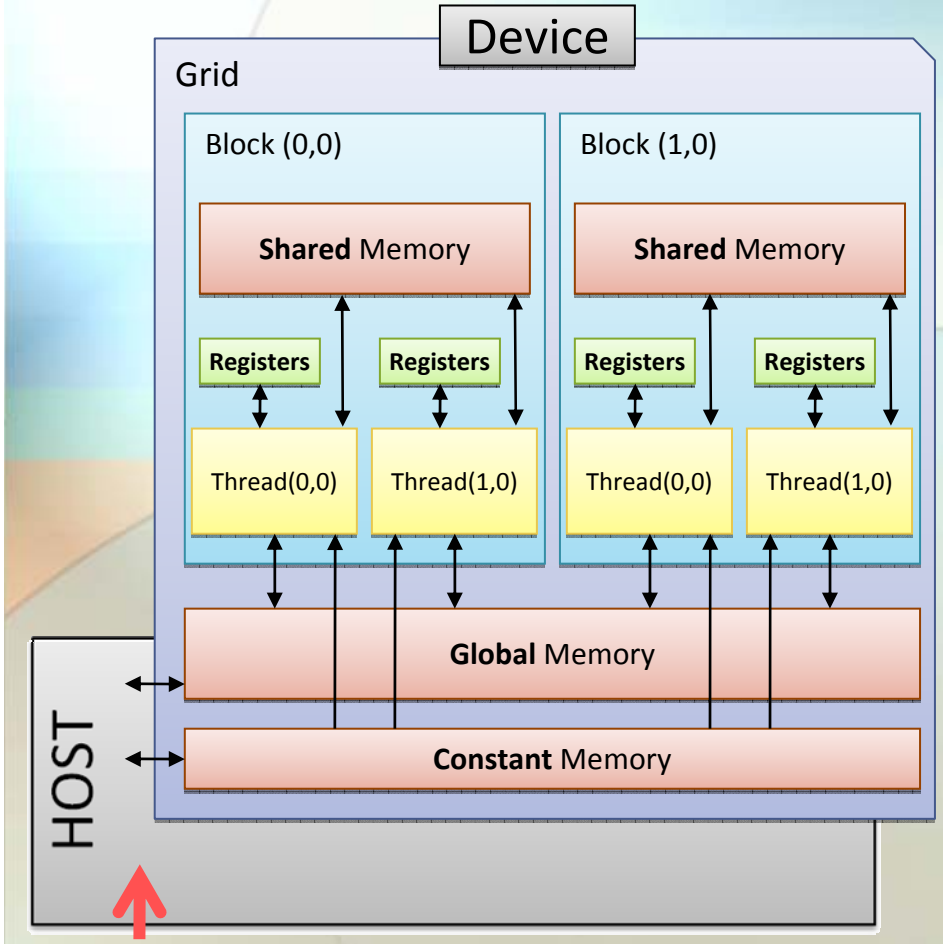
- In order to speedup our kernel, we have to maximize CGMA.

Memory speed

- Tesla C1060 has different access speed for each different memory type.
- Top speed is about 100 GB/s. (cudaMemcpyDeviceToDevice)
- Our program is slower.

- With CGMA ratio of 1.0
  we are below the limit of 25 GB/s.

CUDA memory types



Host:device data transfer

Note: texture memory is not shown here.

- We have seweral types of memory available

Memory type	R	W
Global memory	G	G
Shared memory	B	B
Local memory	T	T
Register memory	T	T
Constant memory	G	

- G = grid
- B = block
- T = thread

R = read, W = write access

Programming CUDA variable type qualifiers

- In our program we can use each memory type.

Variable declaration in C	Memory type	Variable Scope	Memory lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__, __shared__, int sharedVar	Shared	Block	Kernel
__device__, int globalVar	Global	Grid	Application
__device__, __constant__, int constVar	Constant	Grid	Application

- Shared memory declaration: __shared__ int a;

Device variables

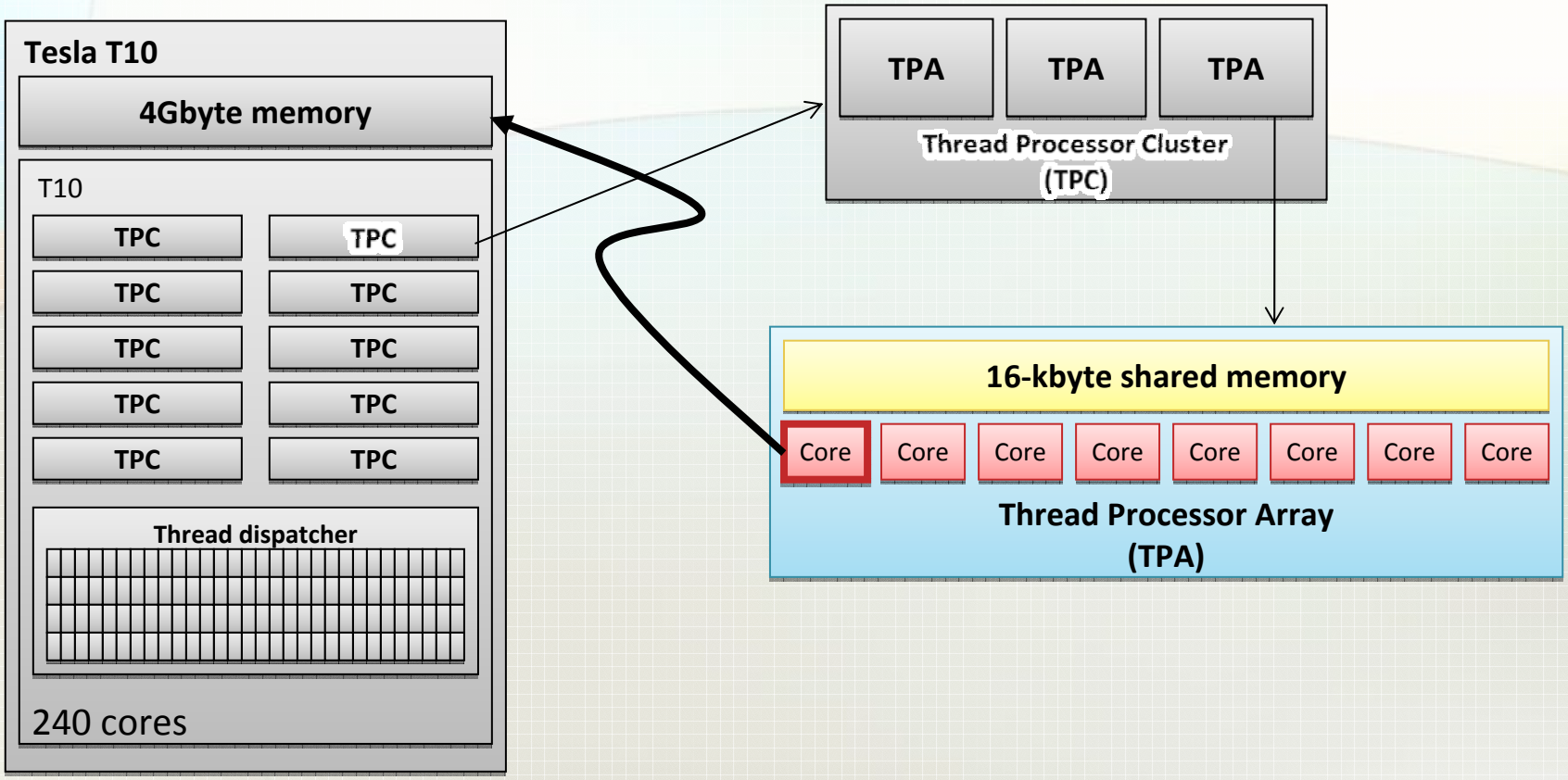
__device__

Host variables

__host__

Type of memory used for now

- Our program is using global GPU memory.

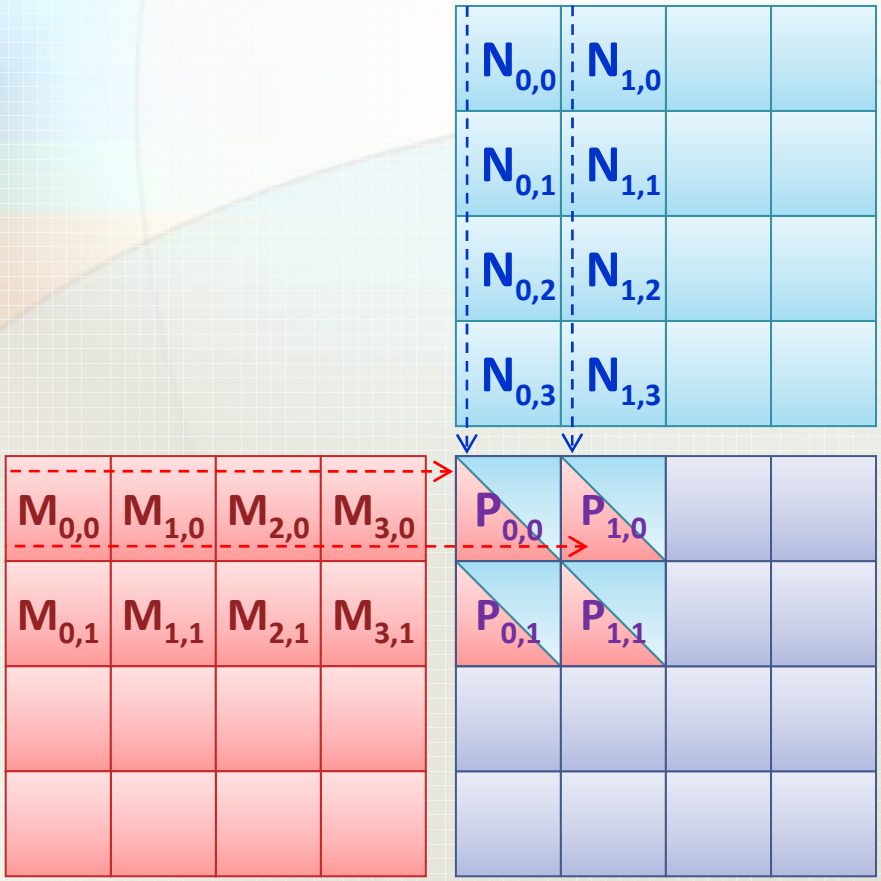


Programming Optimization

- Our program is using slow global GPU memory.
- Now we would like to reduce global memory traffic and increase CGMA.
- We have to analyze our program and detect unnecessary memory operations.

Program analysis

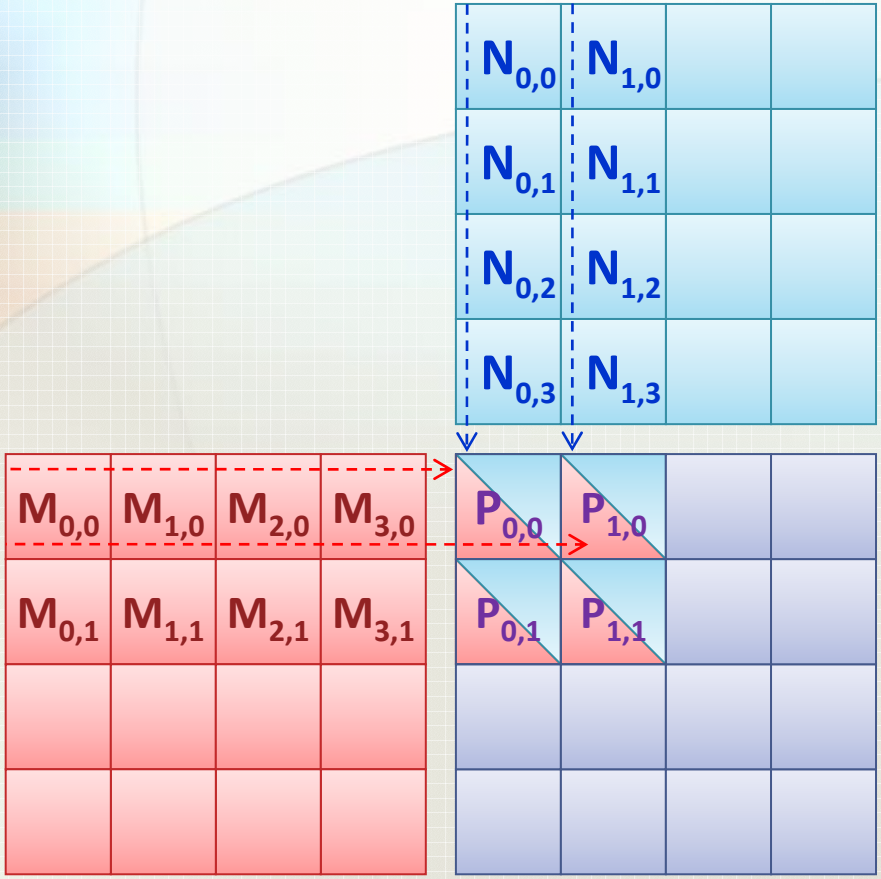
- Lets look how our program is accessing data.



P [0,0]	P [1,0]	P [0,1]	P [1,1]
$M_{0,0}*N_{0,0}$	$M_{0,0}*N_{1,0}$	$M_{0,1}*N_{0,0}$	$M_{0,1}*N_{1,0}$
$M_{1,0}*N_{0,1}$	$M_{1,0}*N_{1,1}$	$M_{1,1}*N_{0,1}$	$M_{1,1}*N_{1,1}$
$M_{2,0}*N_{0,2}$	$M_{2,0}*N_{1,2}$	$M_{2,1}*N_{0,2}$	$M_{2,1}*N_{1,2}$
$M_{3,0}*N_{0,3}$	$M_{3,0}*N_{1,3}$	$M_{3,1}*N_{0,3}$	$M_{3,1}*N_{1,3}$

Program analysis

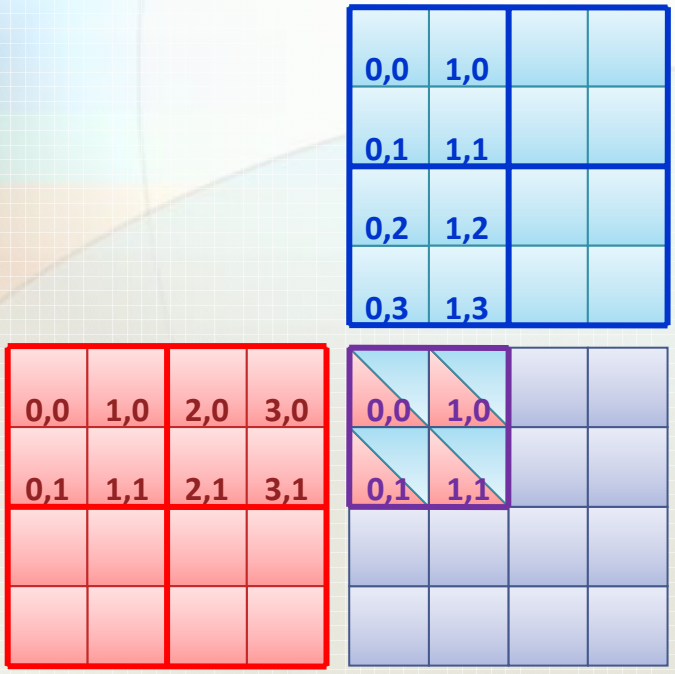
- Some of memory accesses are repeated.



P [0,0]	P [1,0]	P [0,1]	P [1,1]
$M_{0,0}*N_{0,0}$	$M_{0,0}*N_{1,0}$	$M_{0,1}*N_{0,0}$	$M_{0,1}*N_{1,0}$
$M_{1,0}*N_{0,1}$	$M_{1,0}*N_{1,1}$	$M_{1,1}*N_{0,1}$	$M_{1,1}*N_{1,1}$
$M_{2,0}*N_{0,2}$	$M_{2,0}*N_{1,2}$	$M_{2,1}*N_{0,2}$	$M_{2,1}*N_{1,2}$
$M_{3,0}*N_{0,3}$	$M_{3,0}*N_{1,3}$	$M_{3,1}*N_{0,3}$	$M_{3,1}*N_{1,3}$

Memory access optimization

- Now our strategy is to access shared data in phases.



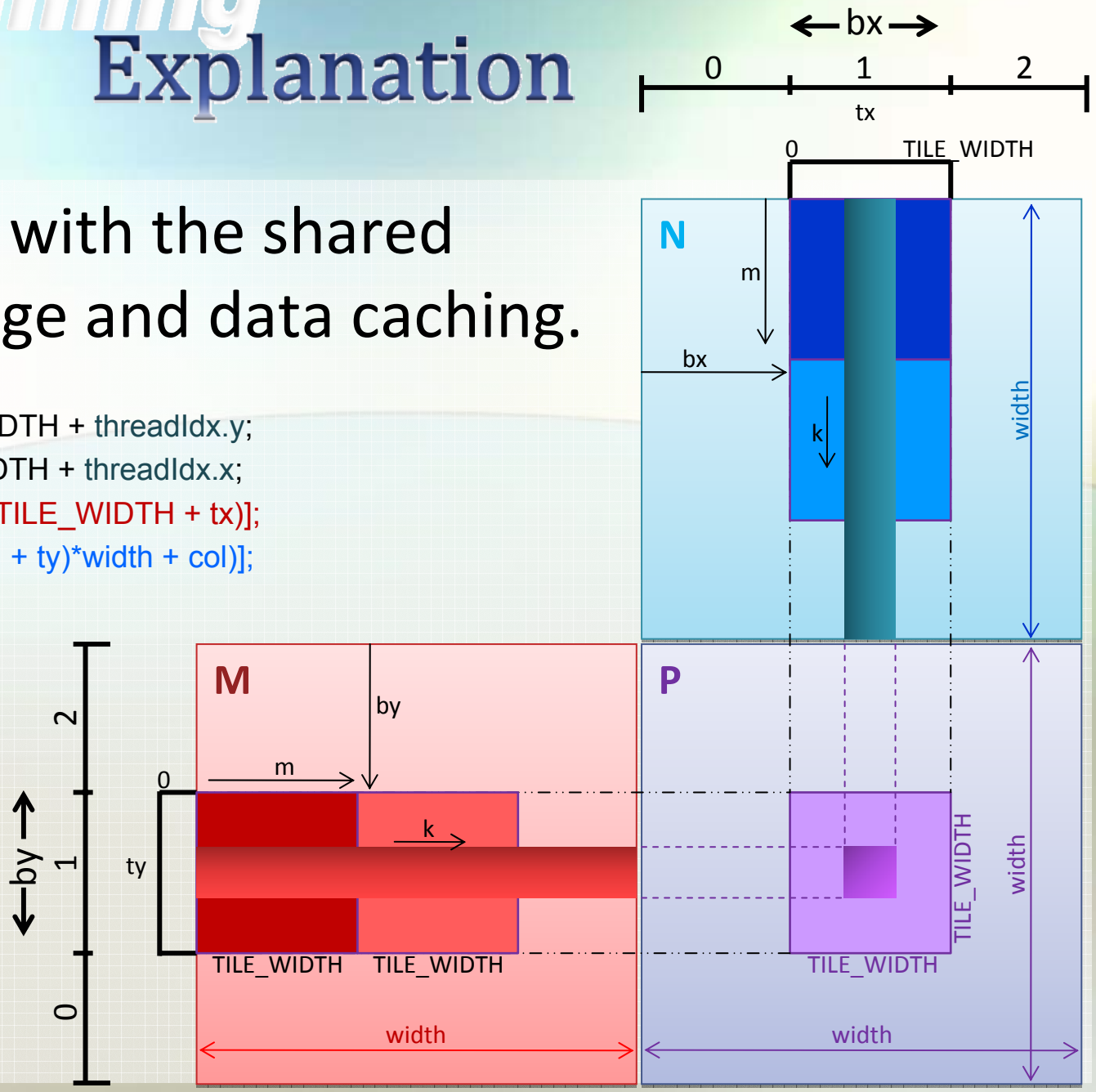
Threads	Phase 1			Phase 2		
$T_{0,0}$	$M_{0,0}$ ↓ $Ms_{0,0}$	$N_{0,0}$ ↓ $Ns_{0,0}$	$Sum_{0,0}$+= $Ms_{0,0}$*$Ns_{0,0}$+ $Ms_{1,0}$*$Ns_{0,1}$	$M_{2,0}$ ↓ $Ms_{0,0}$	$N_{0,2}$ ↓ $Ns_{0,0}$	$Sum_{0,0}$+= $Ms_{0,0}$*$Ns_{0,0}$+ $Ms_{1,0}$*$Ns_{0,1}$
$T_{1,0}$	$M_{1,0}$ ↓ $Ms_{1,0}$	$N_{1,0}$ ↓ $Ns_{1,0}$	$Sum_{1,0}$+= $Ms_{0,0}$*$Ns_{1,0}$+ $Ms_{1,0}$*$Ns_{1,1}$	$M_{3,0}$ ↓ $Ms_{1,0}$	$N_{1,2}$ ↓ $Ns_{1,0}$	$Sum_{1,0}$+= $Ms_{0,0}$*$Ns_{1,0}$+ $Ms_{1,0}$*$Ns_{1,1}$
$T_{0,1}$	$M_{0,1}$ ↓ $Ms_{0,1}$	$N_{0,1}$ ↓ $Ns_{0,1}$	$Sum_{0,1}$+= $Ms_{0,1}$*$Ns_{0,0}$+ $Ms_{1,1}$*$Ns_{0,1}$	$M_{2,1}$ ↓ $Ms_{0,1}$	$N_{0,3}$ ↓ $Ns_{0,1}$	$Sum_{0,1}$+= $Ms_{0,1}$*$Ns_{0,0}$+ $Ms_{1,1}$*$Ns_{0,1}$
$T_{1,1}$	$M_{1,1}$ ↓ $Ms_{1,1}$	$N_{1,1}$ ↓ $Ns_{1,1}$	$Sum_{1,1}$+= $Ms_{0,1}$*$Ns_{1,0}$+ $Ms_{1,1}$*$Ns_{1,1}$	$M_{3,1}$ ↓ $Ms_{1,1}$	$N_{1,3}$ ↓ $Ns_{1,1}$	$Sum_{1,1}$+= $Ms_{0,1}$*$Ns_{1,0}$+ $Ms_{1,1}$*$Ns_{1,1}$

Time

Programming Explanation

- Matrix tiling with the shared memory usage and data caching.

```
int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
int col = blockIdx.x * TILE_WIDTH + threadIdx.x;
Ms[ty][tx] = M[row*width + (m*TILE_WIDTH + tx)];
Ns[ty][tx] = N[(m*TILE_WIDTH + ty)*width + col)];
```

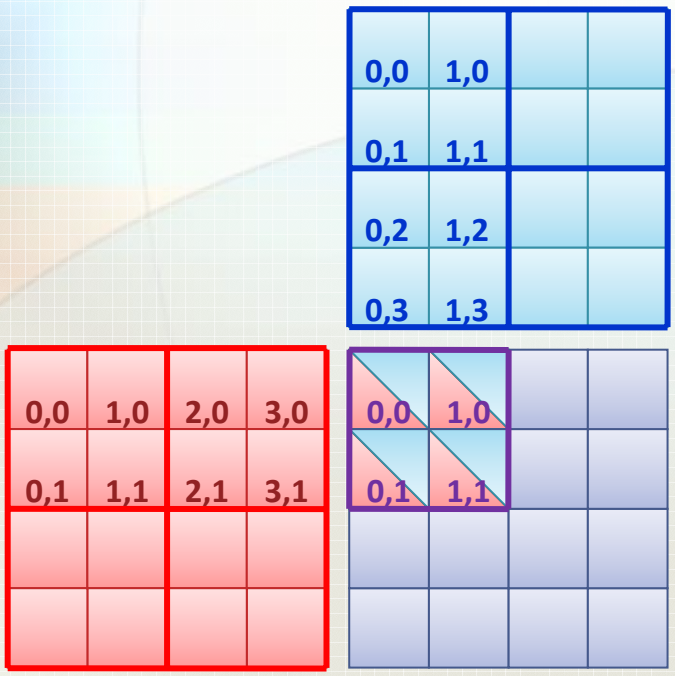


Kernel with shared data usage

```
__global__ void sharedMul( float* P, float* M, float* N, int width ) {

  __shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Ns[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;  int tx = threadIdx.x; int ty = threadIdx.y;

  int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

  float sum = 0.0;       // element computed by the thread
  for (int m=0; m < width/TILE_WIDTH; m++) {
    Ms[ty][tx] = M[row*width + (m*TILE_WIDTH + tx)];
    Ns[ty][tx] = N[(m*TILE_WIDTH + ty)*width + col)];
    __syncthreads();

    for (k = 0; k < width; k++)
      sum += M[row*width+k] * N[k*width+col];
    __syncthreads();
  }

  P[row*width+col] = sum;  // element [tx,ty] is now calculated
}

/* run the kernel */
dim3 blockDim( width/TILE_WIDTH, width/TILE_WIDTH );
dim3 gridDim(TILE_WIDTH,TILE_WIDTH);

sharedMul <<< gridDim, blockDim >>> ( P, M, N, width );
```

Memory access optimization

- Now our strategy is to access shared data in phases.



Threads	Phase 1			Phase 2		
$T_{0,0}$	$M_{0,0}$ ↓ $Ms_{0,0}$	$N_{0,0}$ ↓ $Ns_{0,0}$	$Sum_{0,0}$+= $Ms_{0,0}$*$Ns_{0,0}$+ $Ms_{1,0}$*$Ns_{0,1}$	$M_{2,0}$ ↓ $Ms_{0,0}$	$N_{0,2}$ ↓ $Ns_{0,0}$	$Sum_{0,0}$+= $Ms_{0,0}$*$Ns_{0,0}$+ $Ms_{1,0}$*$Ns_{0,1}$
$T_{1,0}$	$M_{1,0}$ ↓ $Ms_{1,0}$	$N_{1,0}$ ↓ $Ns_{1,0}$	$Sum_{1,0}$+= $Ms_{0,0}$*$Ns_{1,0}$+ $Ms_{1,0}$*$Ns_{1,1}$	$M_{3,0}$ ↓ $Ms_{1,0}$	$N_{1,2}$ ↓ $Ns_{1,0}$	$Sum_{1,0}$+= $Ms_{0,0}$*$Ns_{1,0}$+ $Ms_{1,0}$*$Ns_{1,1}$
$T_{0,1}$	$M_{0,1}$ ↓ $Ms_{0,1}$	$N_{0,1}$ ↓ $Ns_{0,1}$	$Sum_{0,1}$+= $Ms_{0,1}$*$Ns_{0,0}$+ $Ms_{1,1}$*$Ns_{0,1}$	$M_{2,1}$ ↓ $Ms_{0,1}$	$N_{0,3}$ ↓ $Ns_{0,1}$	$Sum_{0,1}$+= $Ms_{0,1}$*$Ns_{0,0}$+ $Ms_{1,1}$*$Ns_{0,1}$
$T_{1,1}$	$M_{1,1}$ ↓ $Ms_{1,1}$	$N_{1,1}$ ↓ $Ns_{1,1}$	$Sum_{1,1}$+= $Ms_{0,1}$*$Ns_{1,0}$+ $Ms_{1,1}$*$Ns_{1,1}$	$M_{3,1}$ ↓ $Ms_{1,1}$	$N_{1,3}$ ↓ $Ns_{1,1}$	$Sum_{1,1}$+= $Ms_{0,1}$*$Ns_{1,0}$+ $Ms_{1,1}$*$Ns_{1,1}$

Time

Results comparision

Pros and cons of CUDA programming

SUMMARY AND DISCUSSION

Discussion

Programs comparision

- Some benchmark results of programs

CPU exclusive run		GPU with global memory		GPU with shared memory	
SIZE	TIME [s]	SIZE	TIME [s]	SIZE	TIME [s]
2	9.743286e-08	2	2.896618e-05	2	1.689454e-05
4	1.494033e-07	4	2.955391e-05	4	1.807400e-05
8	7.512598e-07	8	3.000104e-05	8	1.638880e-05
16	5.197445e-06	16	5.449054e-05	16	1.720564e-05
32	3.879064e-05	32	1.006258e-04	32	2.136294e-05
64	3.382112e-04	64	2.920939e-04	64	3.029413e-05
128	3.561758e-03	128	2.319826e-03	128	7.786980e-05
256	3.198436e-02	256	1.673443e-02	256	2.657310e-04
512	3.056566e-01	512	1.333204e-01	512	2.124511e-03
1024	9.015563e+00	1024	1.062235e+00	1024	1.163225e-02
2048	8.564798e+01	2048	8.486859e+00	2048	8.727042e-02
4096	7.524644e+02	4096	6.796577e+01	4096	6.912488e-01

Graph

Discussion

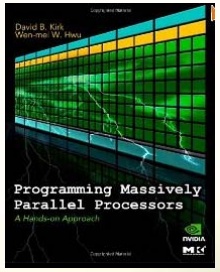
Summary

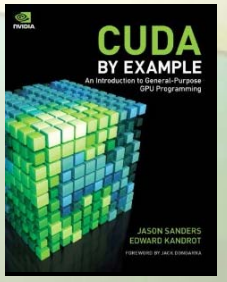
- Pros and cons of CUDA:
  - Massive parallelizm and enormous speedup.
  - New exciting technology for the masses.
  - Ability to latency hiding.
  - Cost effective computational power.
  - Forces to rethink and to rewrite old projects.
  - Low-level programming required.
  - Not standard compliant (OpenCL?)
  - Separated memory space and hardware setup.

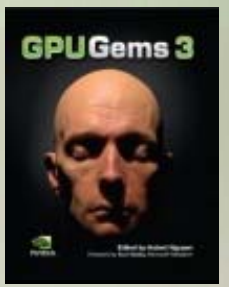
Discussion

References

- Books
  - David B.Kirk & Wen-mei W.Hwu, *Programming Massively Parallel Processors, A Hands-on Approach*, NVIDIA, Elsevier, Morgan Kaufmann, ISBN:978-0-12-381472-2
  - Jason Sanders, Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, This title has not yet been released.
  - Huber Nguyen, *GPU Gems 3*, Addison-Wesley, ISBN:978-0-321-51526-1, this one is also available as free web version: http://developer.nvidia.com/object/gpu-gems-3.html



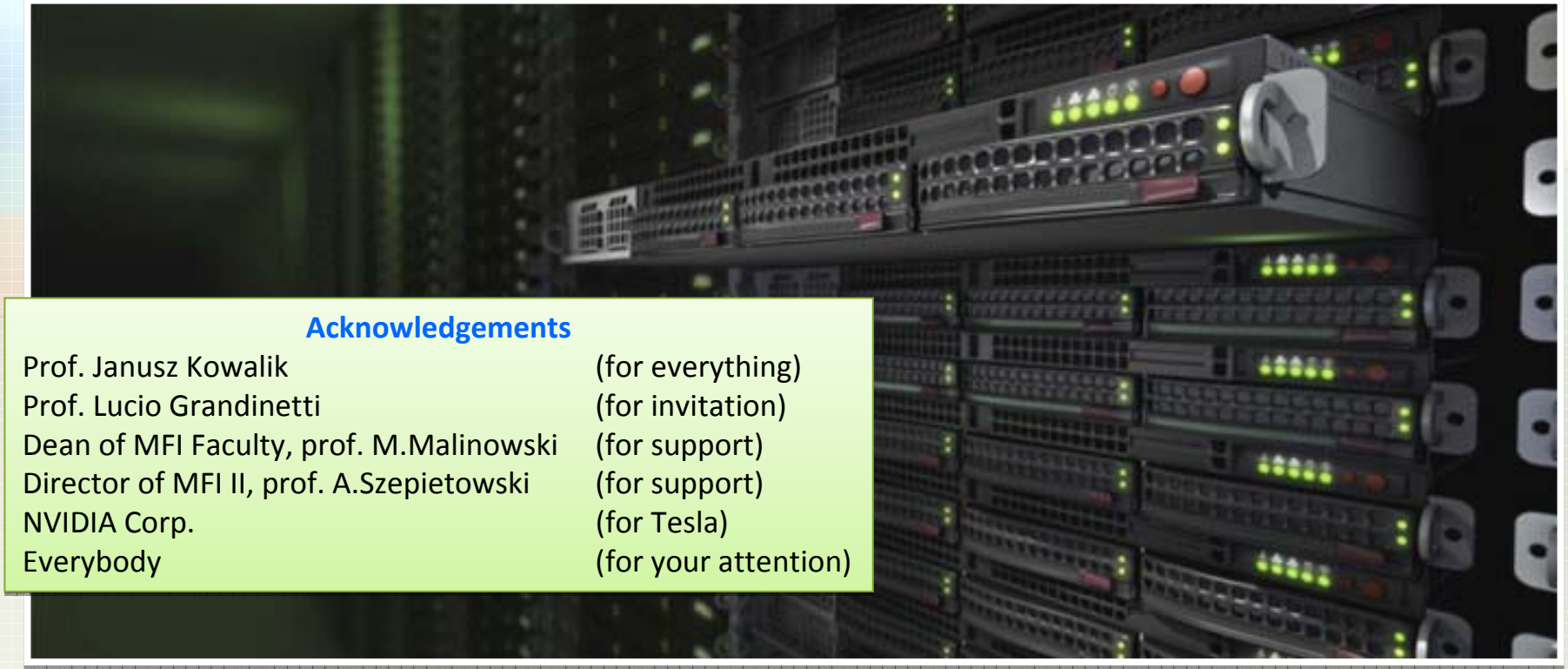




- World Wide Web
  - http://developer.nvidia.com/page/home.html (documentation)
  - http://developer.nvidia.com/object/cuda_training.html (lectures)

The end

- Acknowledgements

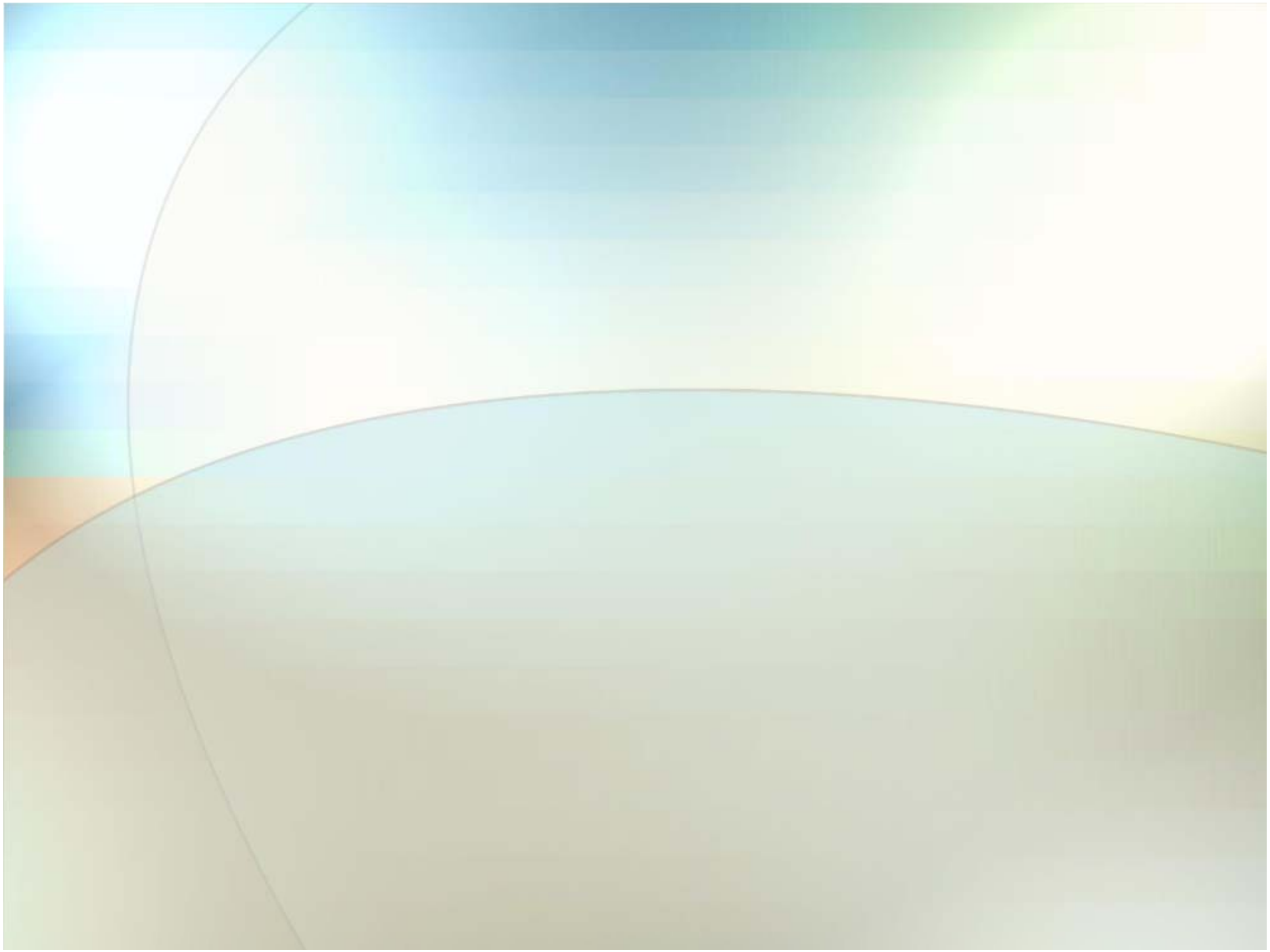


Acknowledgements

Prof. Janusz Kowalik	(for everything)
Prof. Lucio Grandinetti	(for invitation)
Dean of MFI Faculty, prof. M.Malinowski	(for support)
Director of MFI II, prof. A.Szepietowski	(for support)
NVIDIA Corp.	(for Tesla)
Everybody	(for your attention)

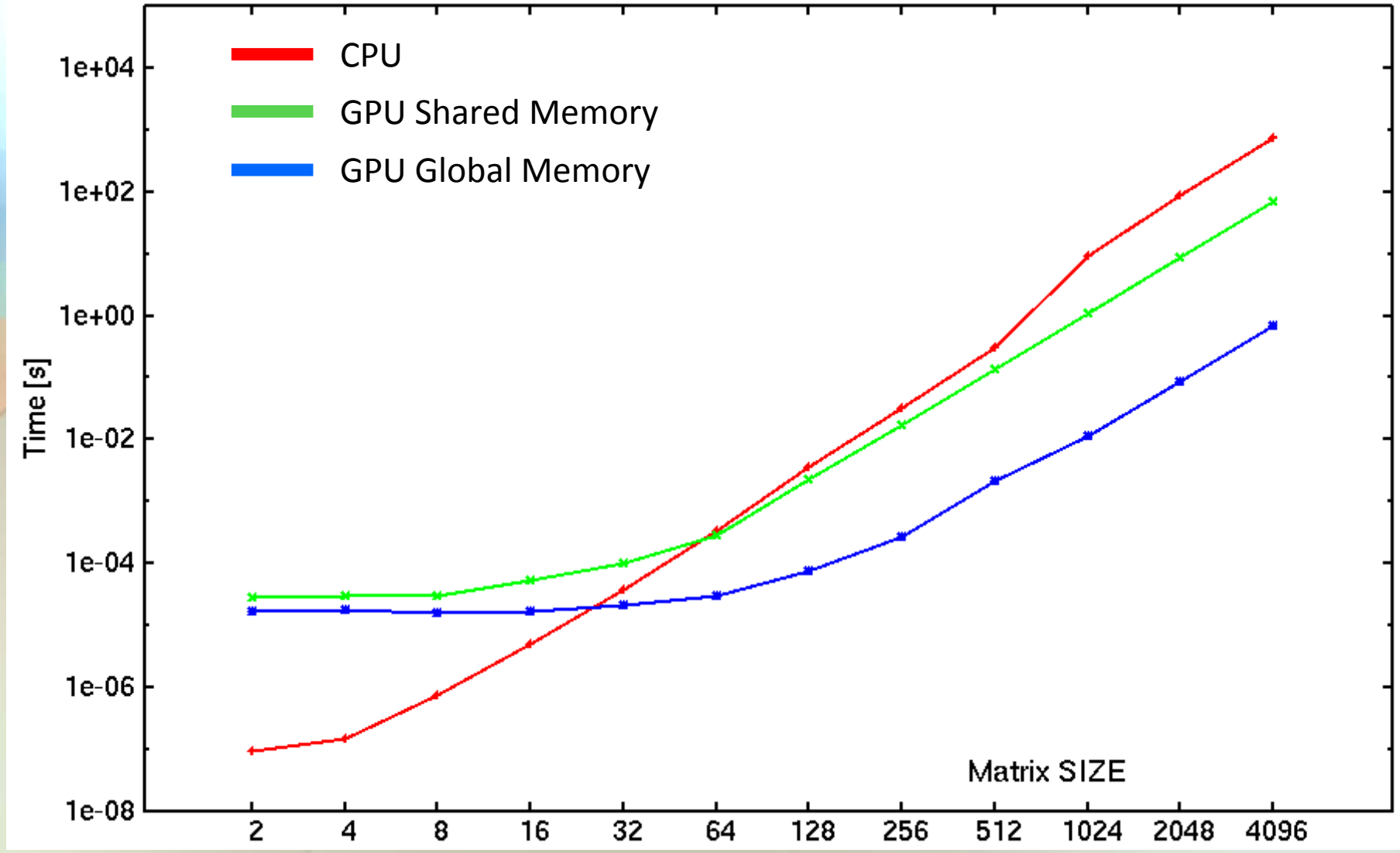
Source: nvidia.com

Thank you!



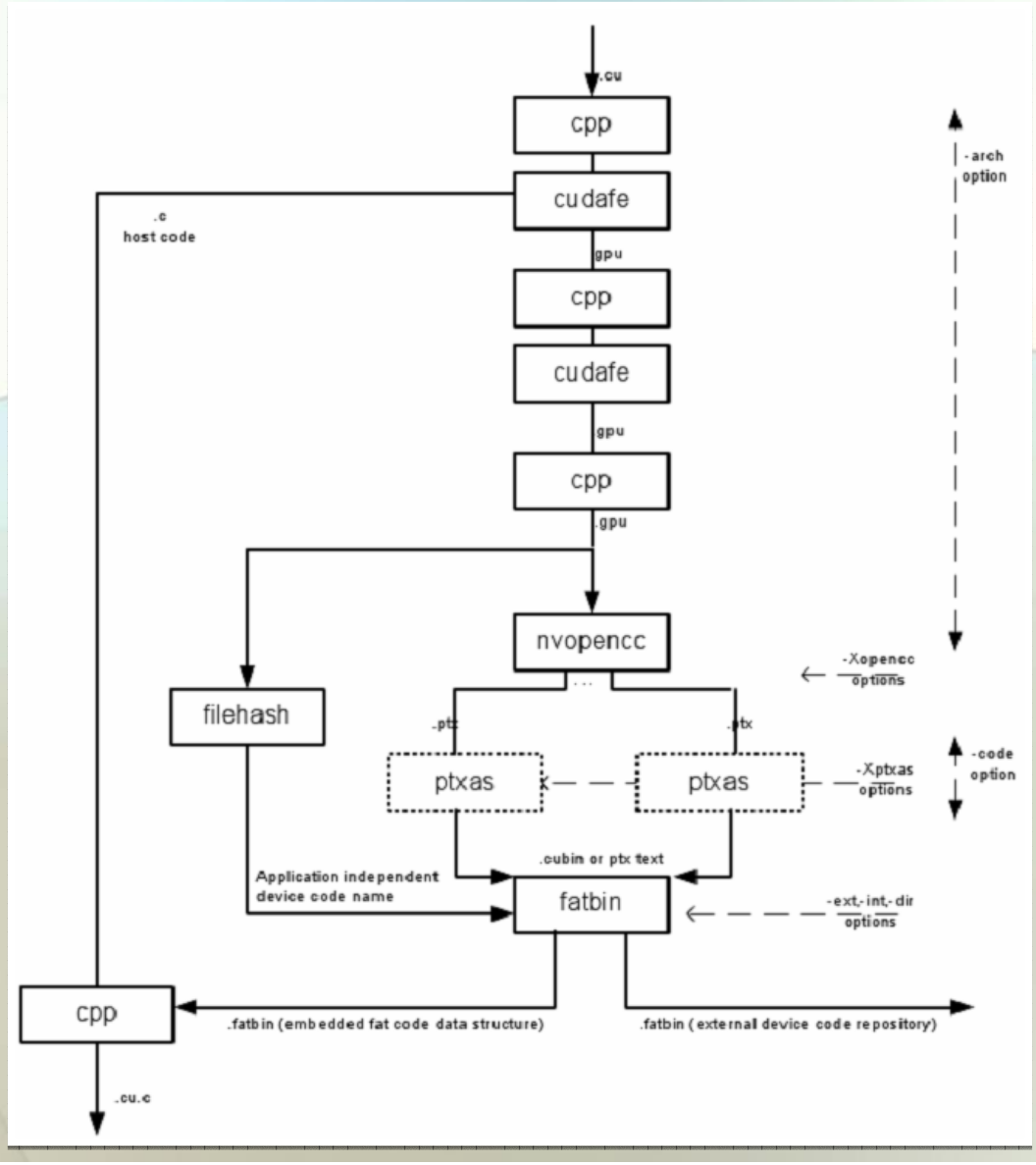
Results comparision

Back



Software

How nvcc works



RETURN

Software

How nvcc works

Return to CUDA Tools

Installing CUDA under Linux

BACK TO INSTALL

- On Fedora, Ubuntu, RedHat, SUSE, Gentoo, etc.:
  - Download proper file (it usually has `.run` extension)
    - `devdriver_3.0_linux_64_195.36.15.run`
  - Run the file as root, using `sh` command:

```
# sudo sh devdriver_3.0_linux_64_195.36.15.run
```

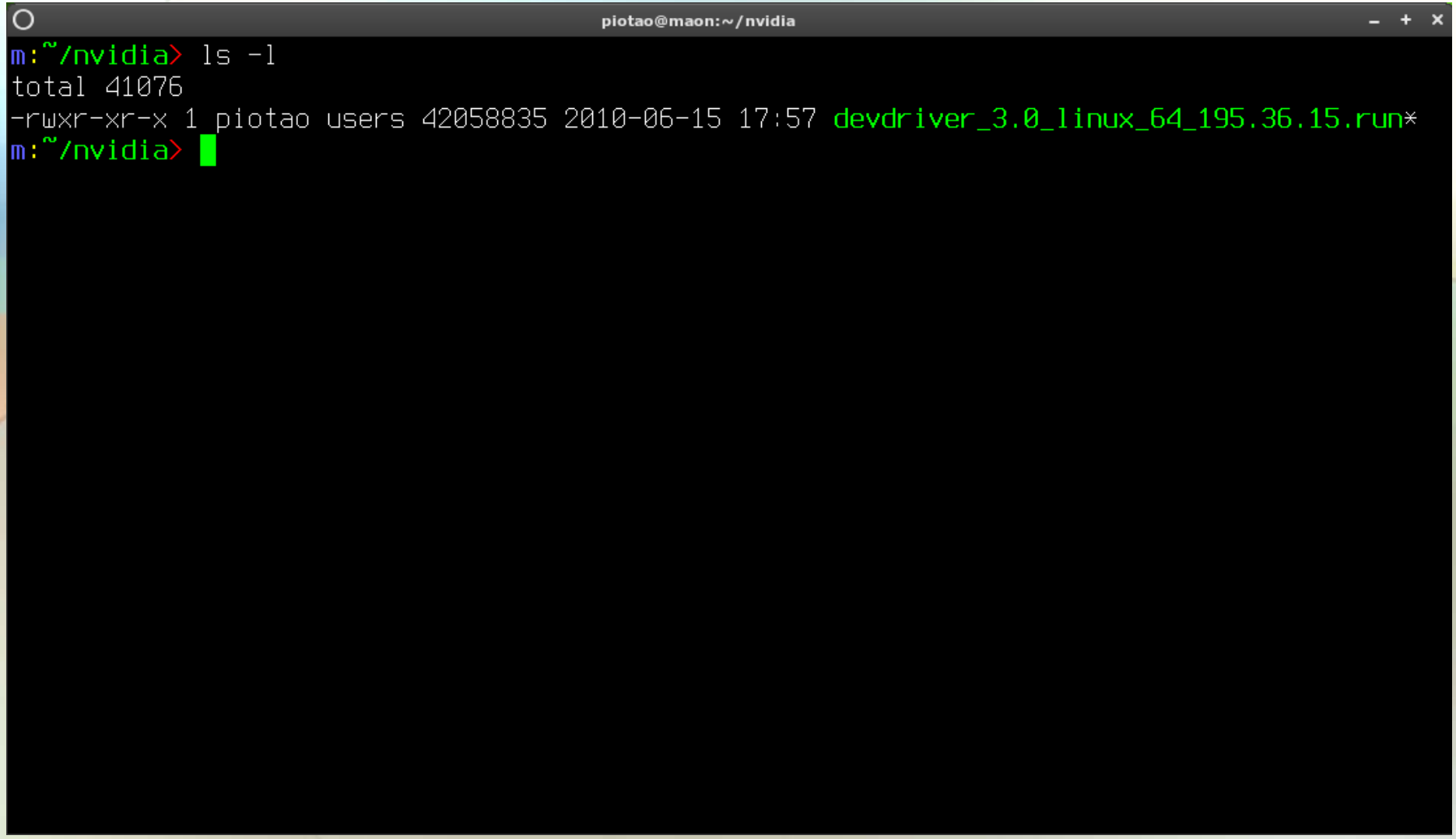
- Verify your install
  - Run the following command:

```
# sudo sh devdriver_3.0_linux_64_195.36.15.run -i
```

Installation

Linux CUDA installation

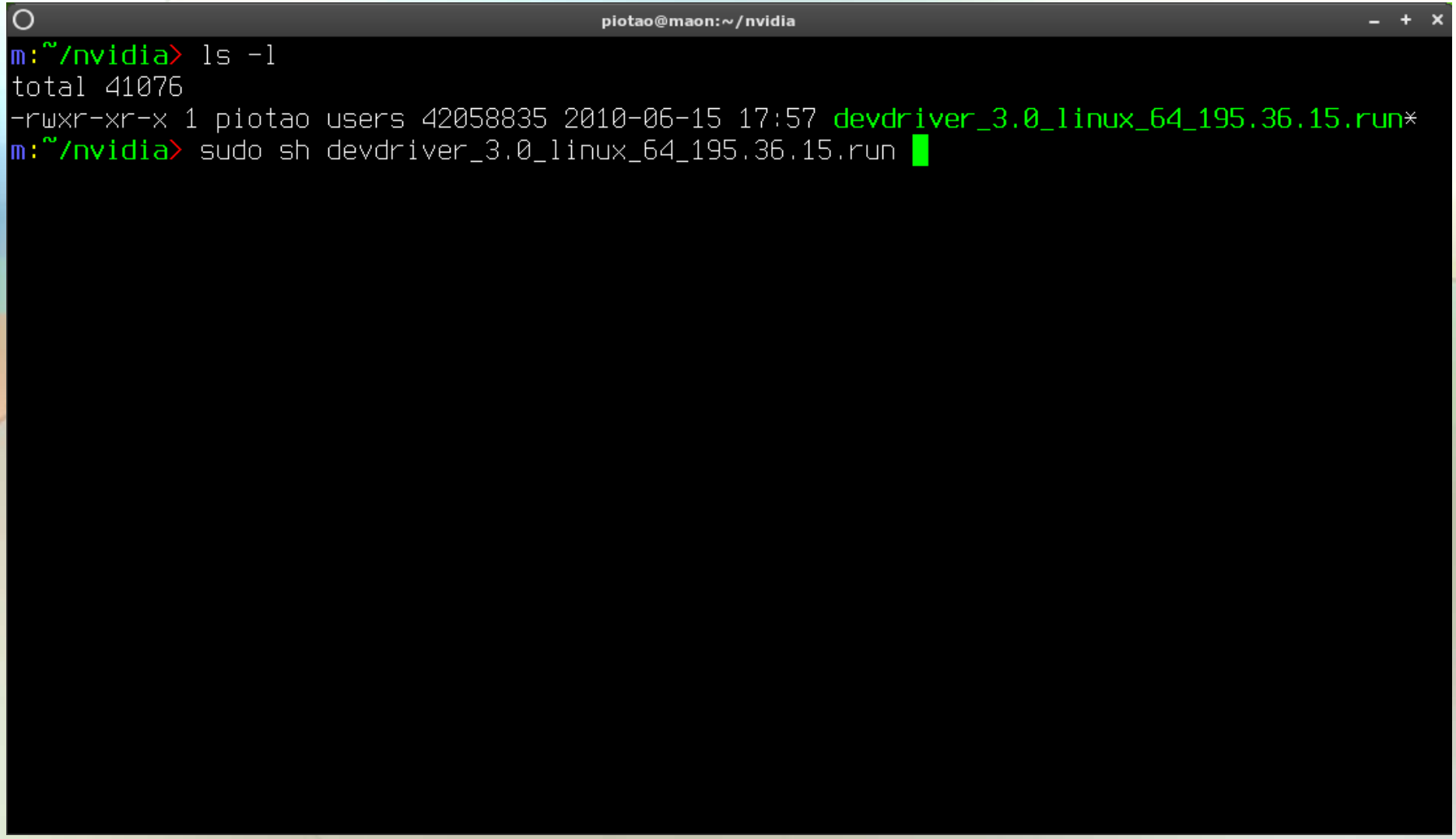
Return



Installation

Linux CUDA installation

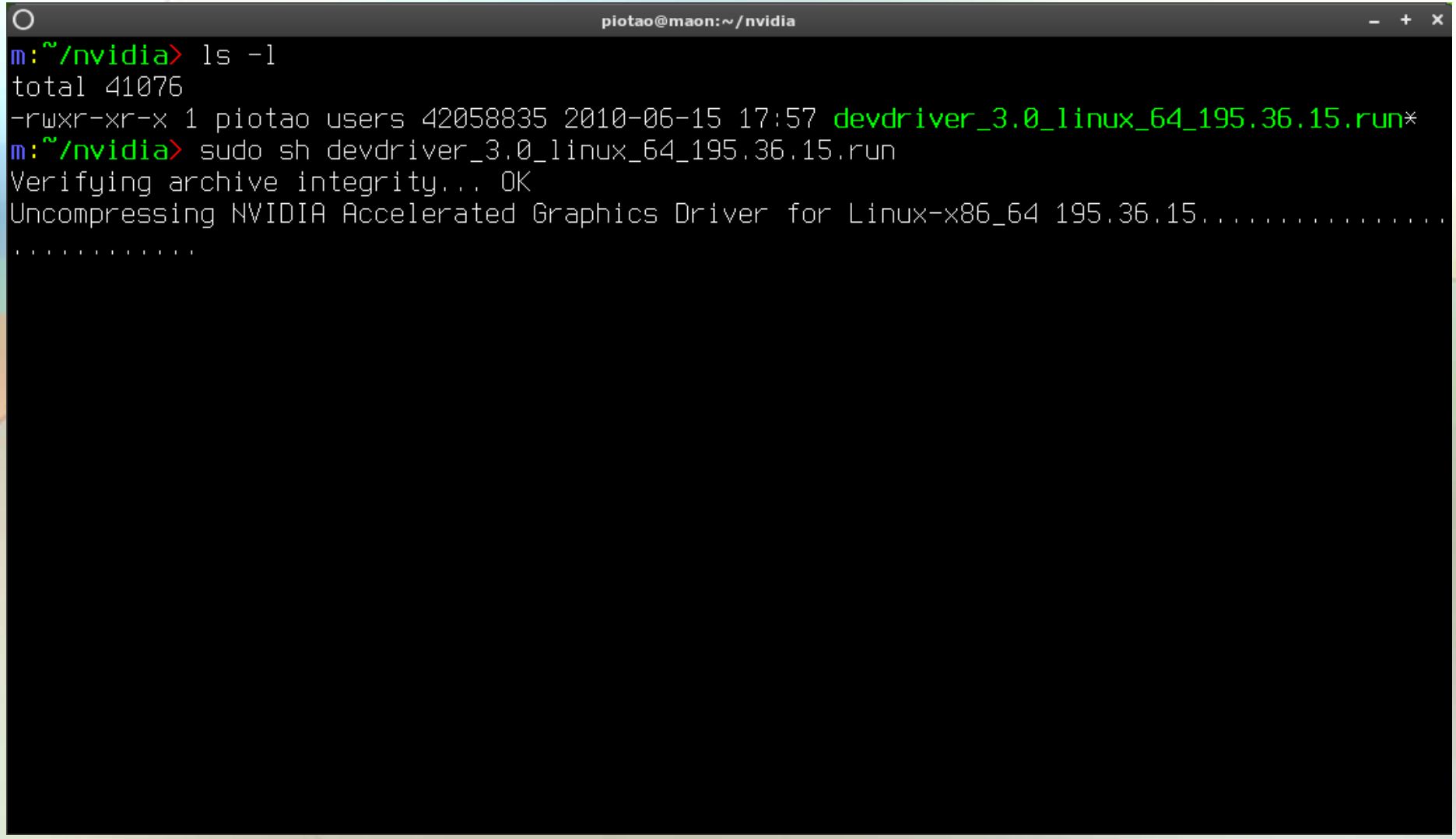
Return



Installation

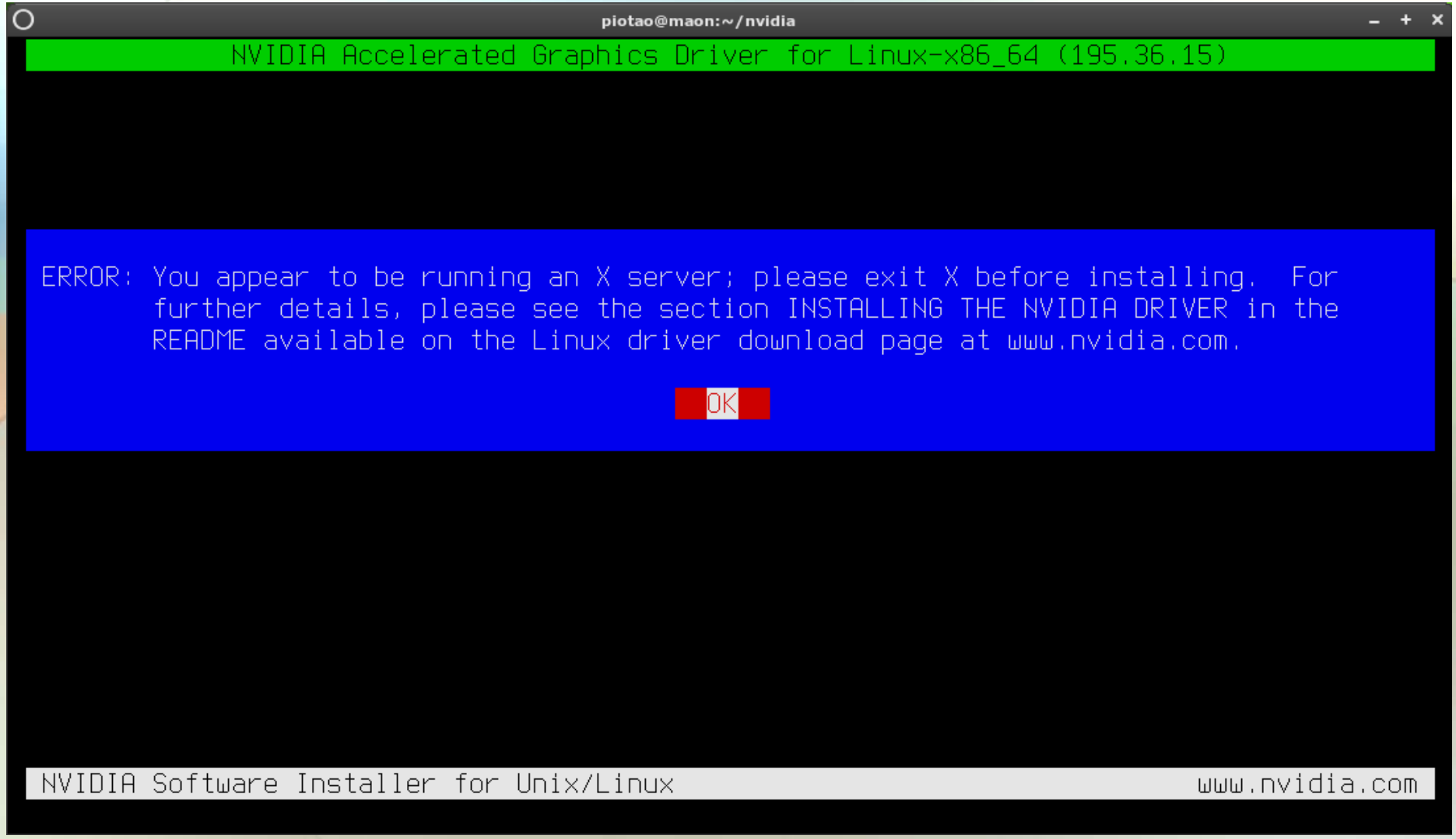
Linux CUDA installation

Return



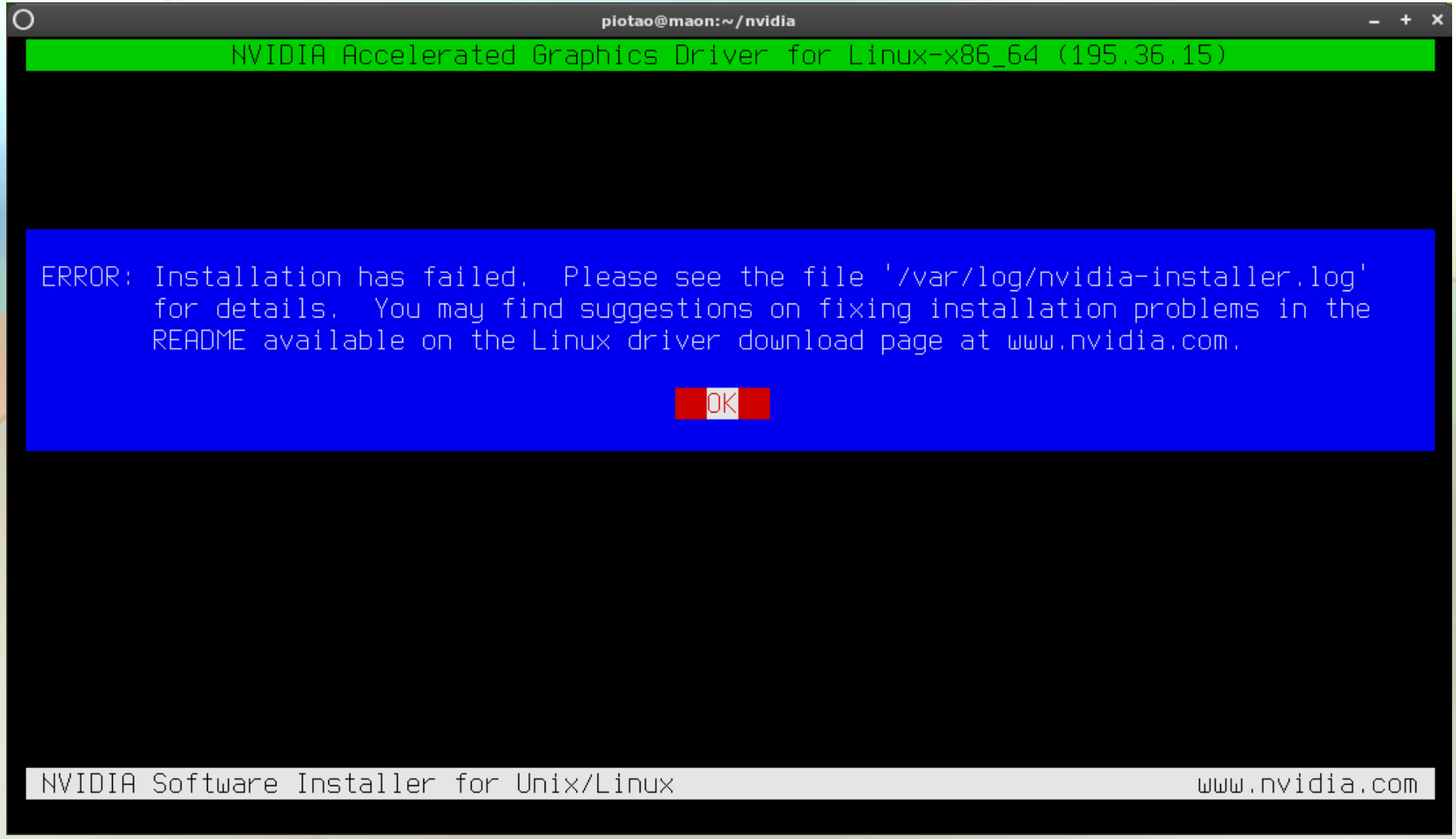
Linux CUDA installation

Return



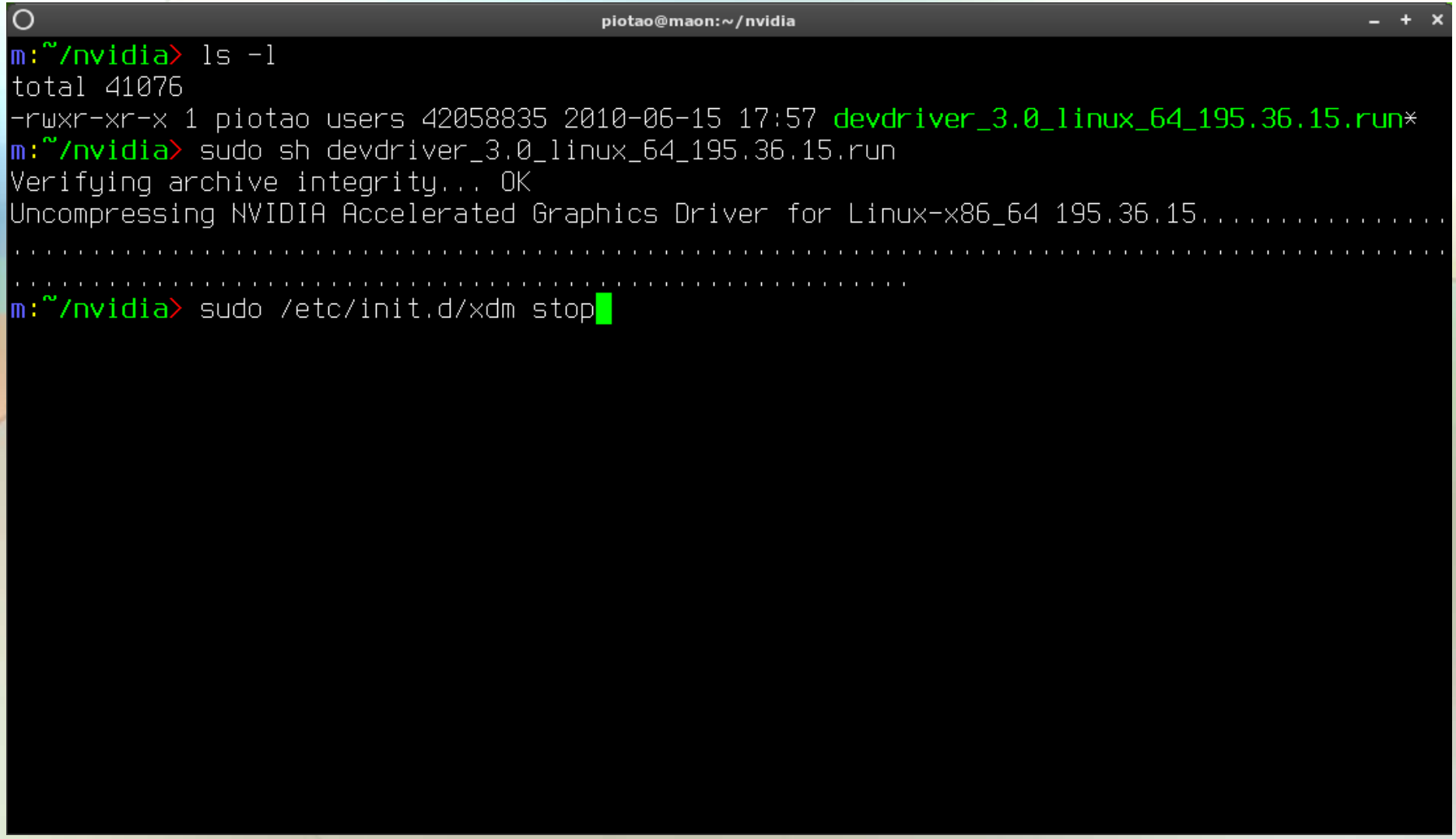
Linux CUDA installation

Return



Linux CUDA installation

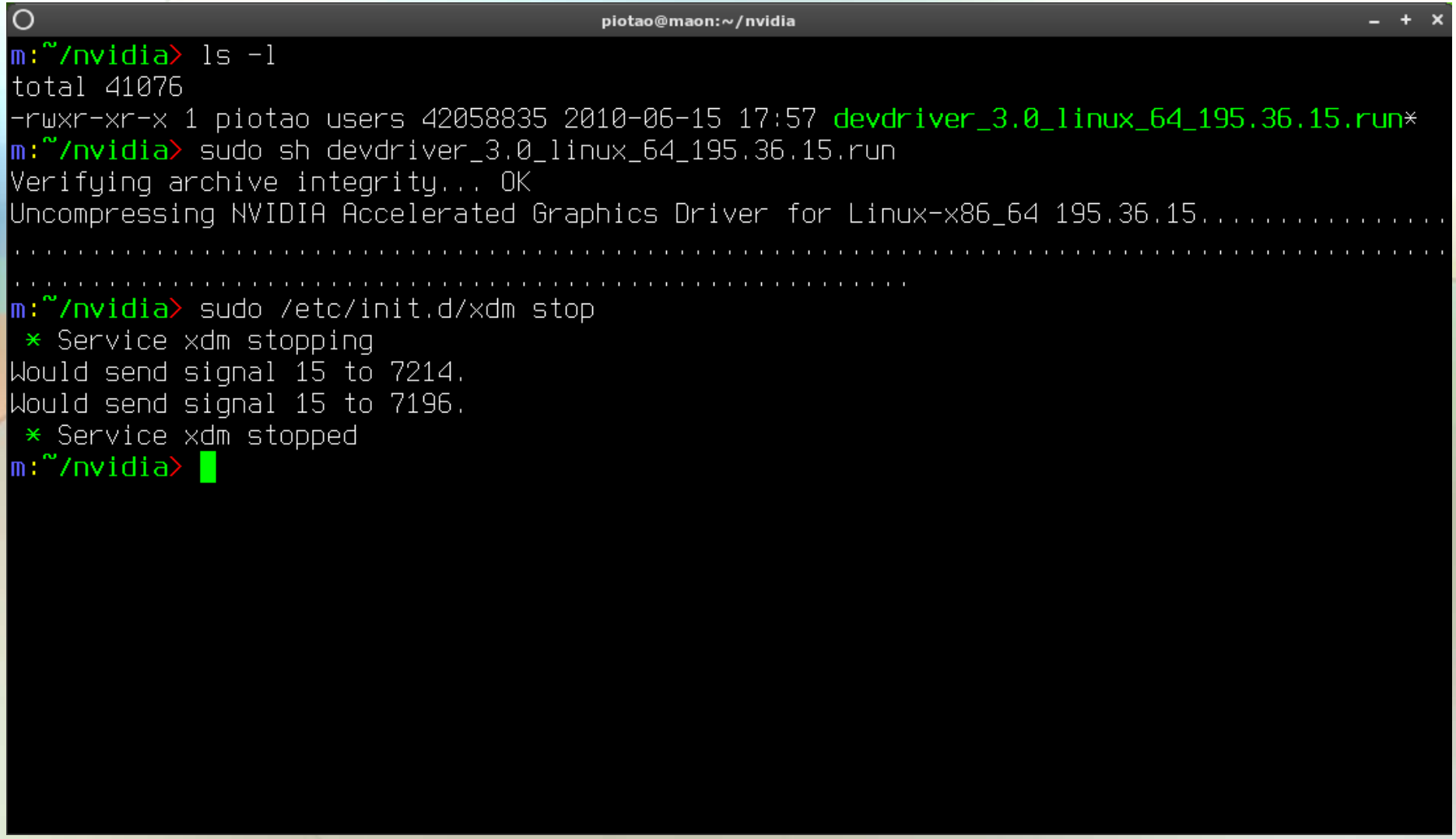
Return



Installation

Linux CUDA installation

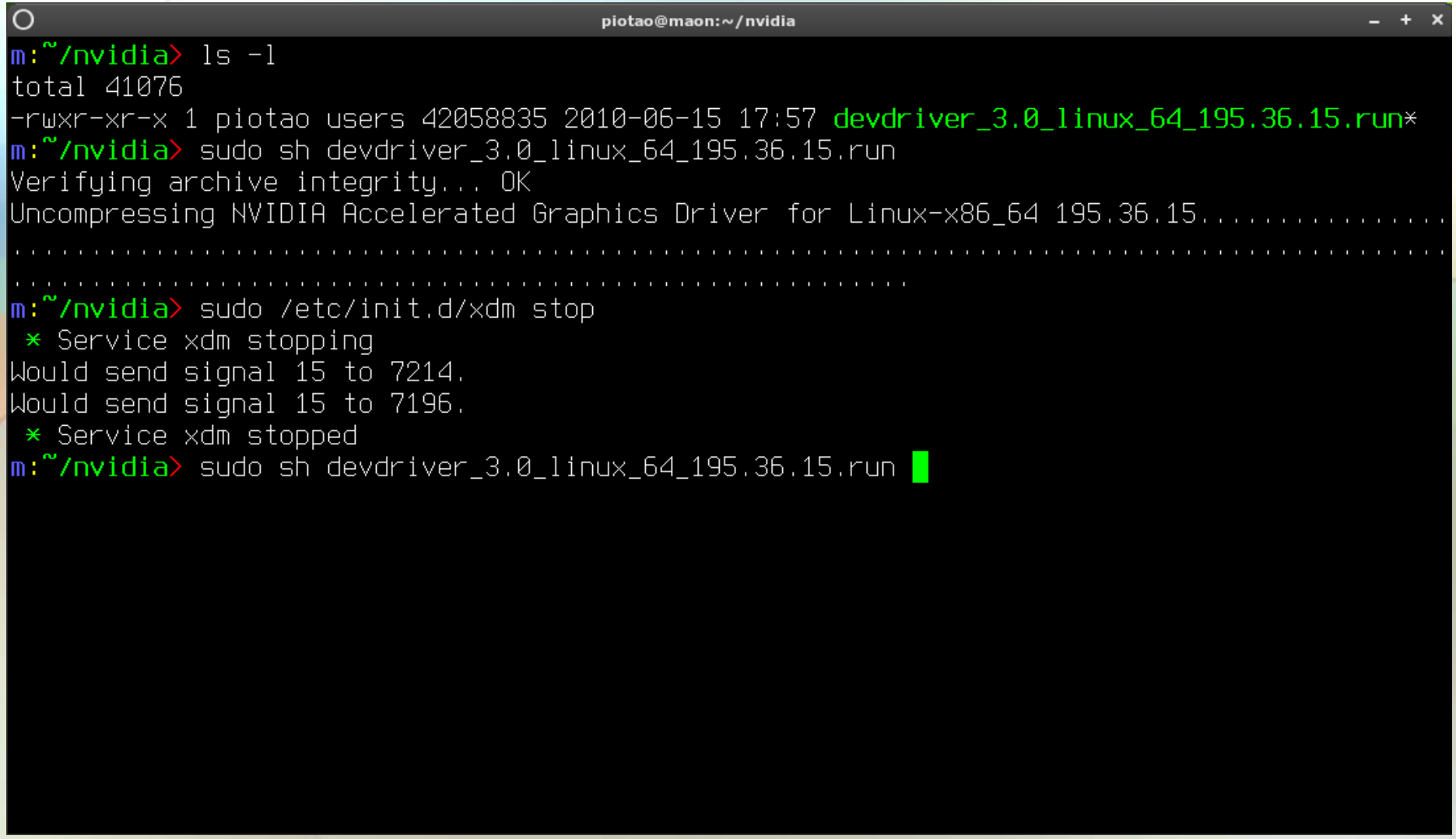
Return



Installation

Linux CUDA installation

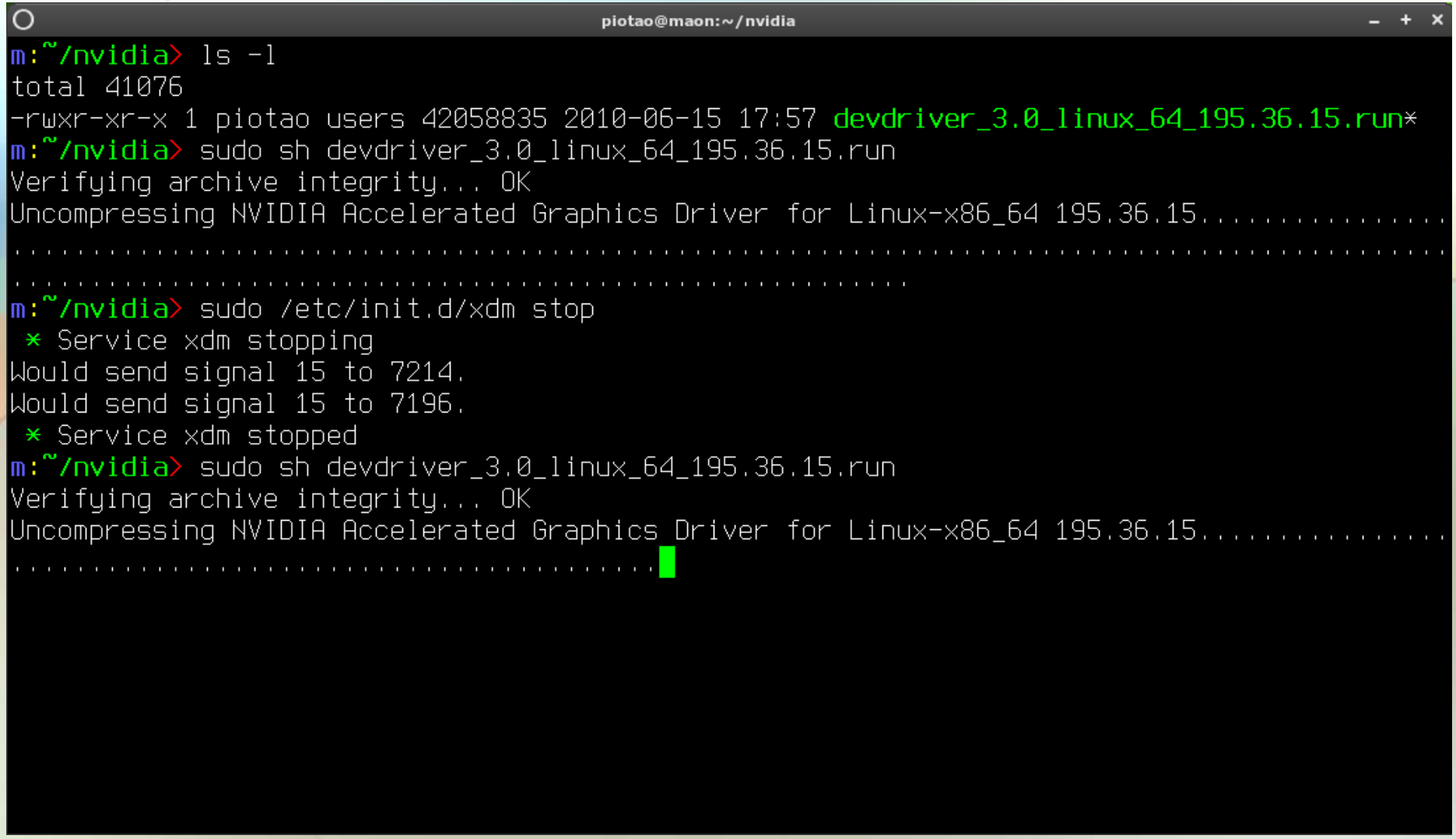
Return



Installation

Linux CUDA installation

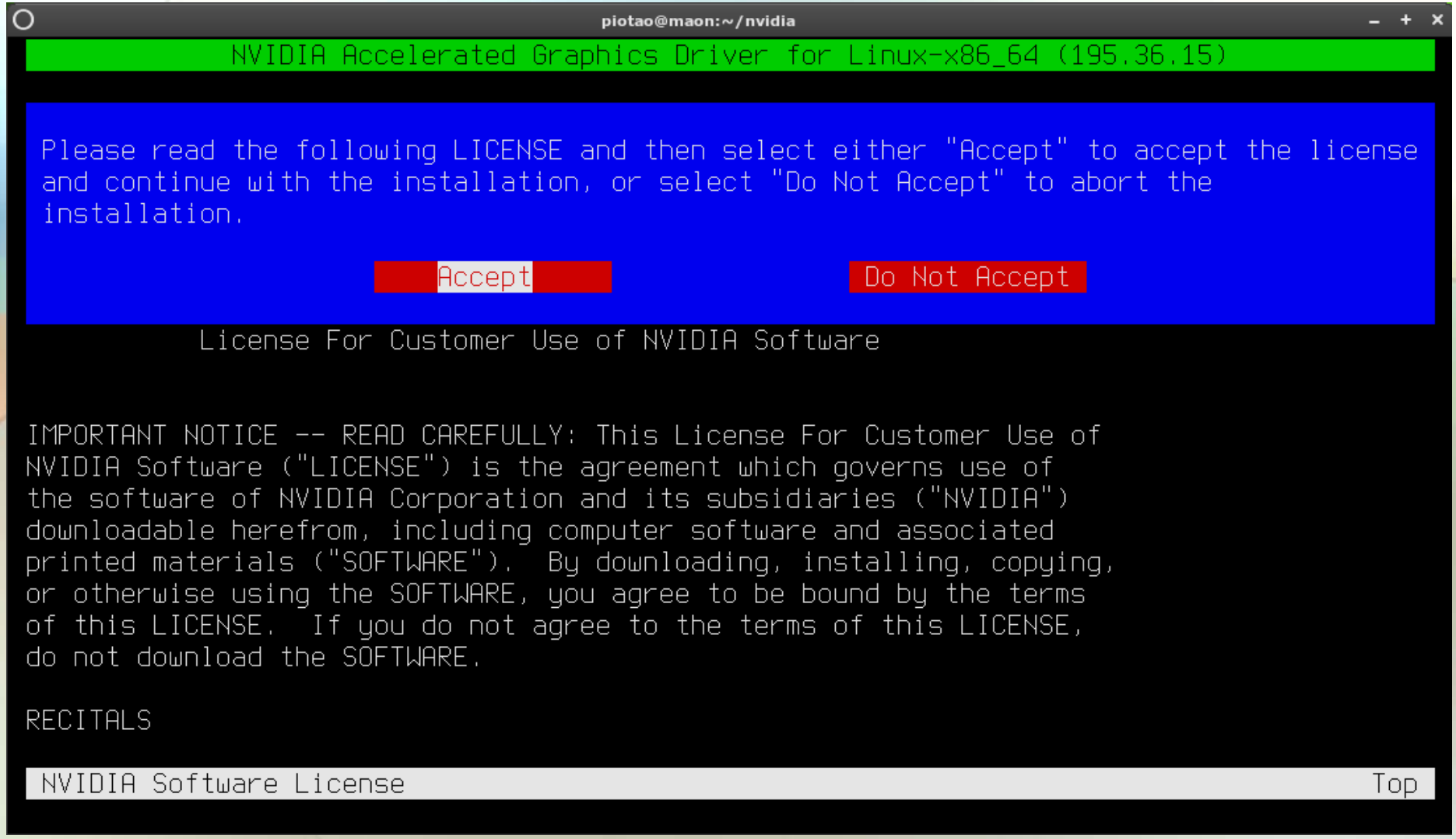
Return



Installation

Linux CUDA installation

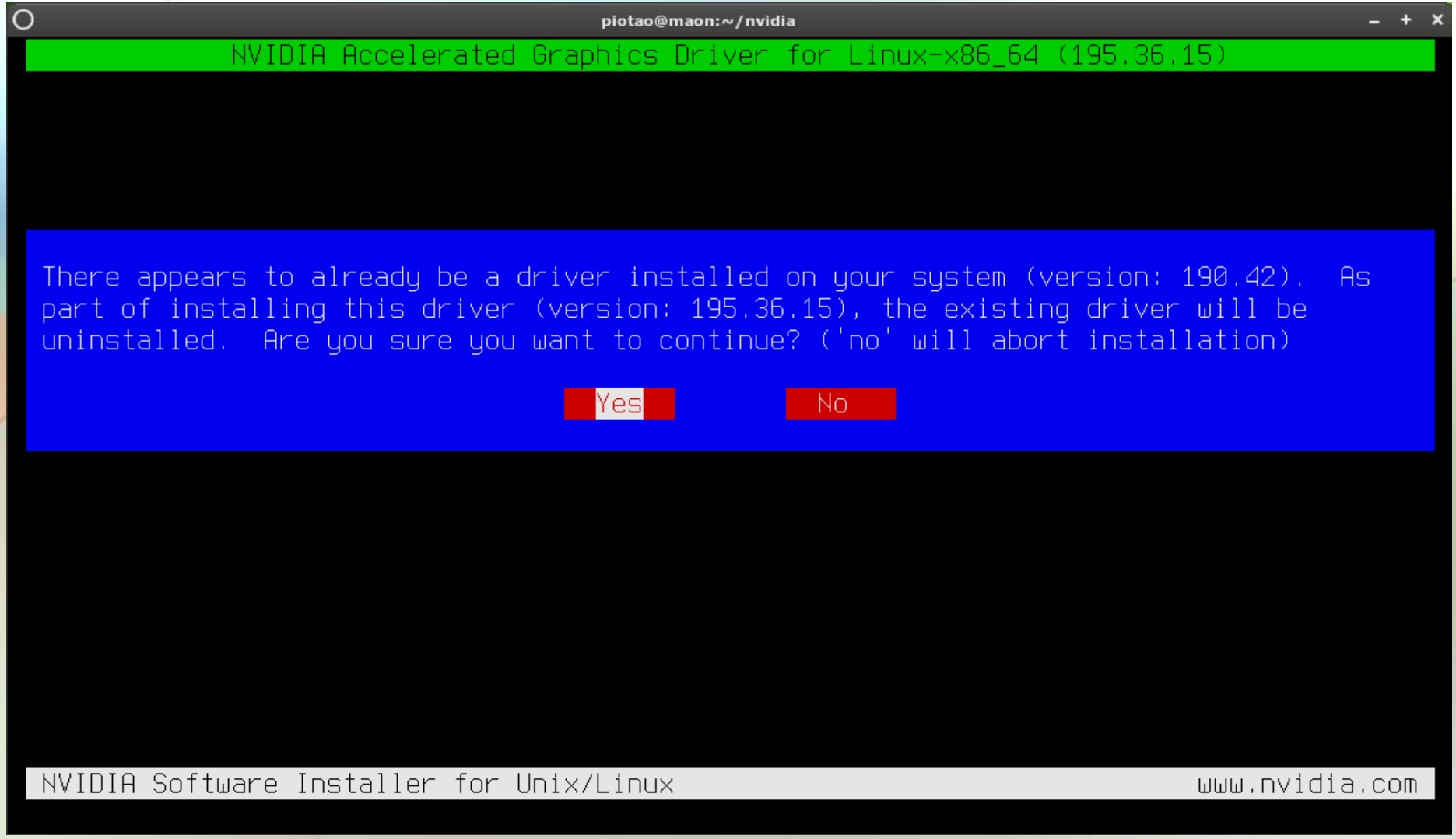
Return



Installation

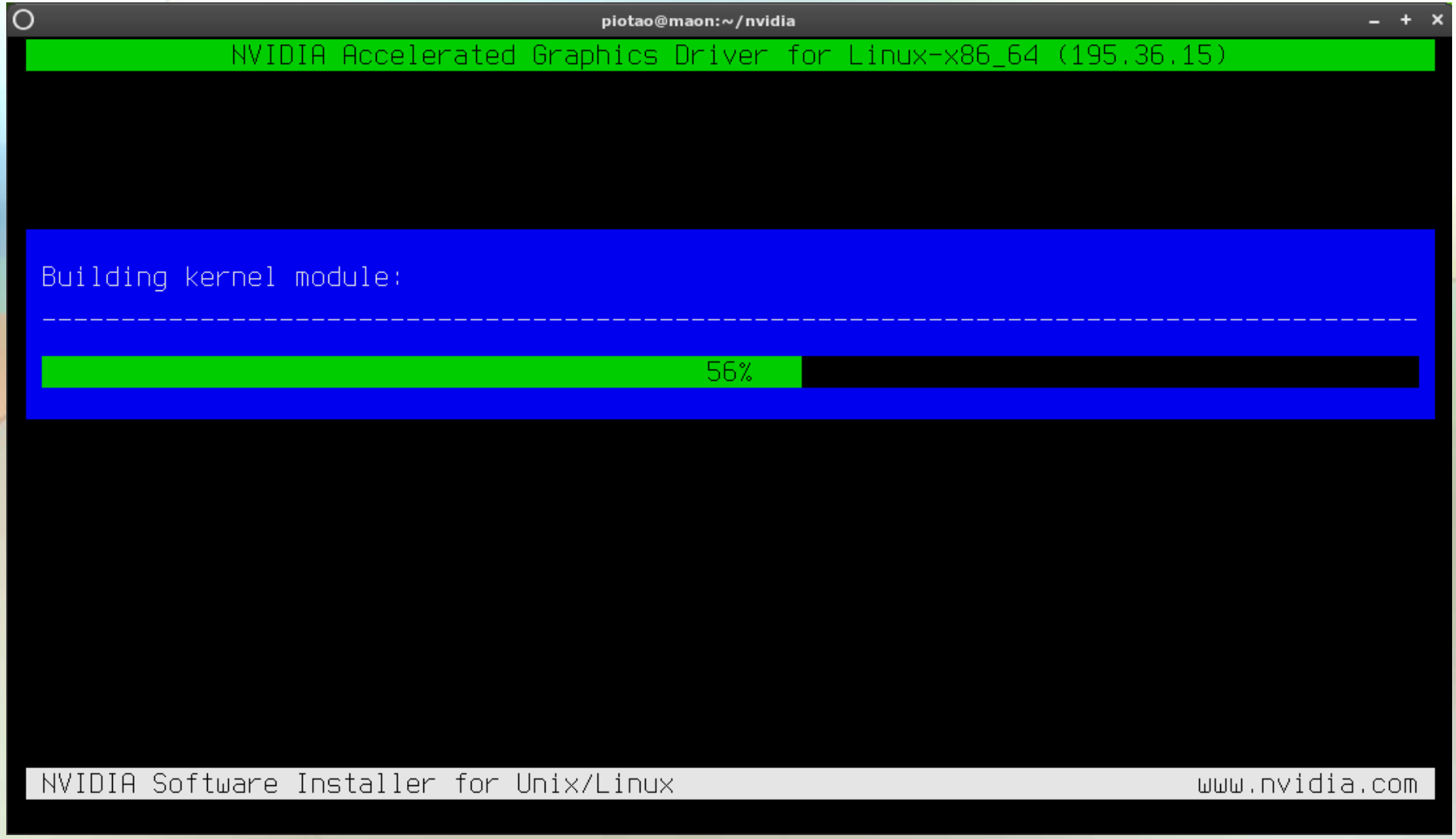
Linux CUDA installation

Return



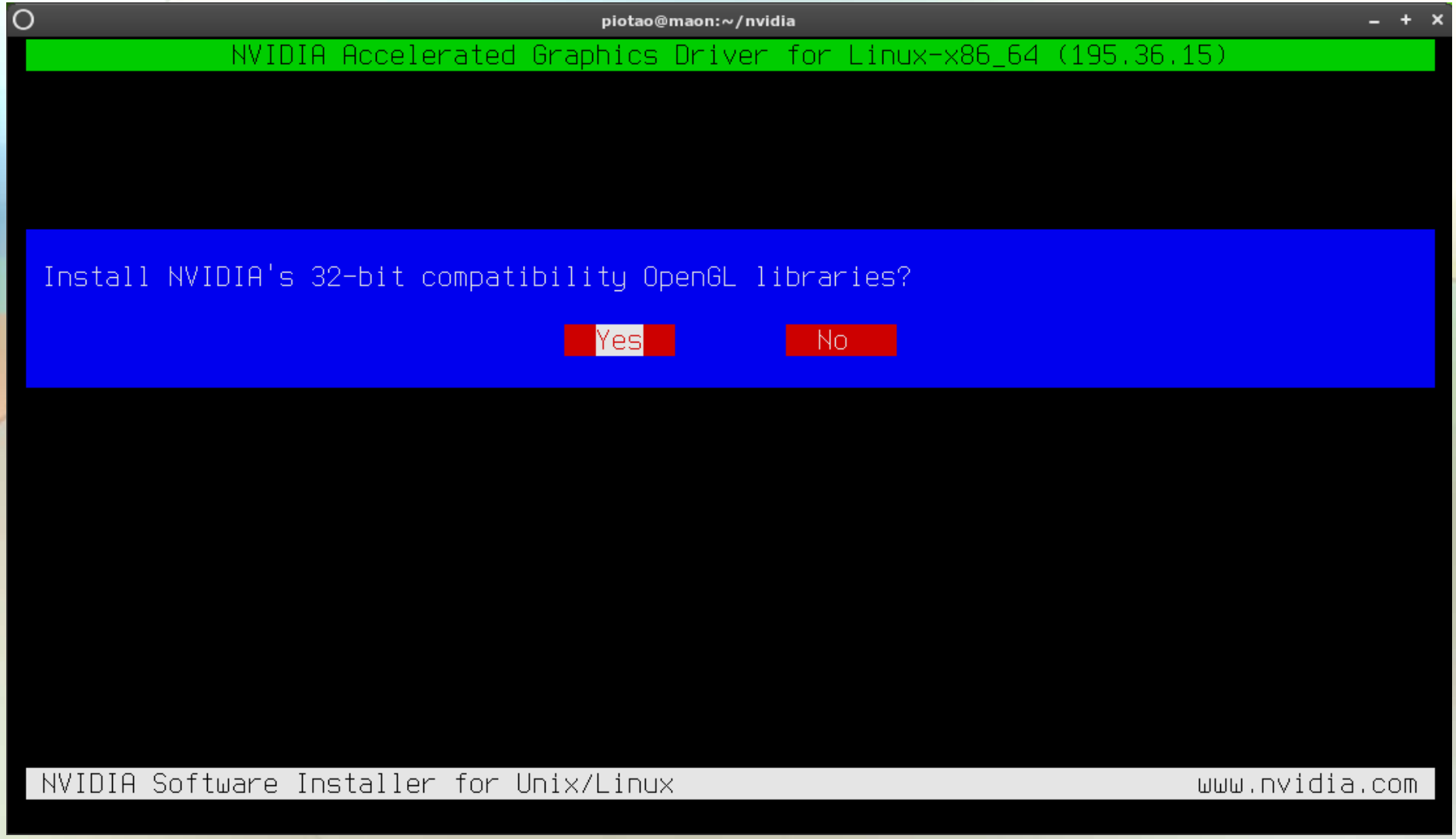
Linux CUDA installation

Return



Linux CUDA installation

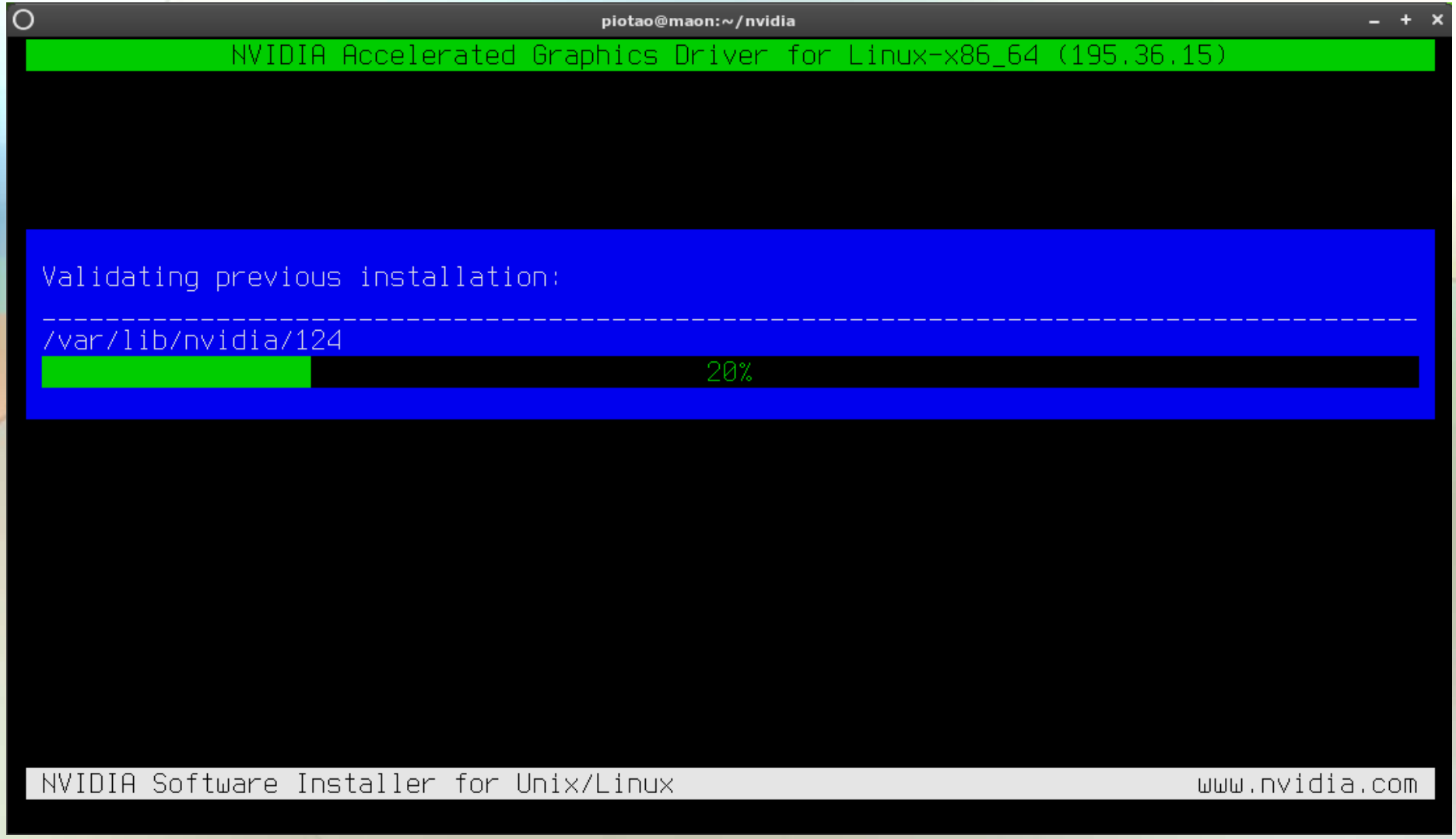
Return



Installation

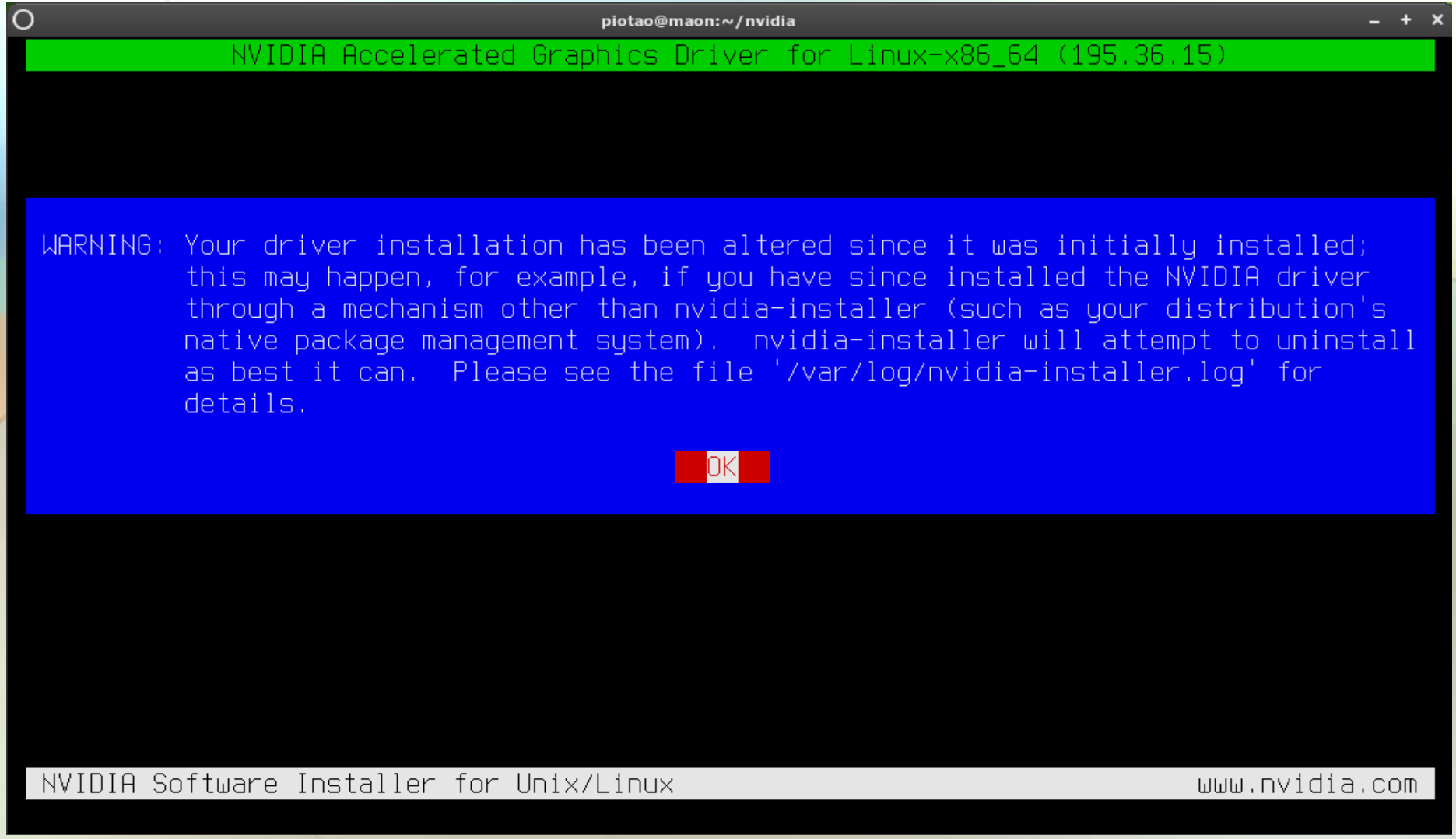
Linux CUDA installation

Return



Linux CUDA installation

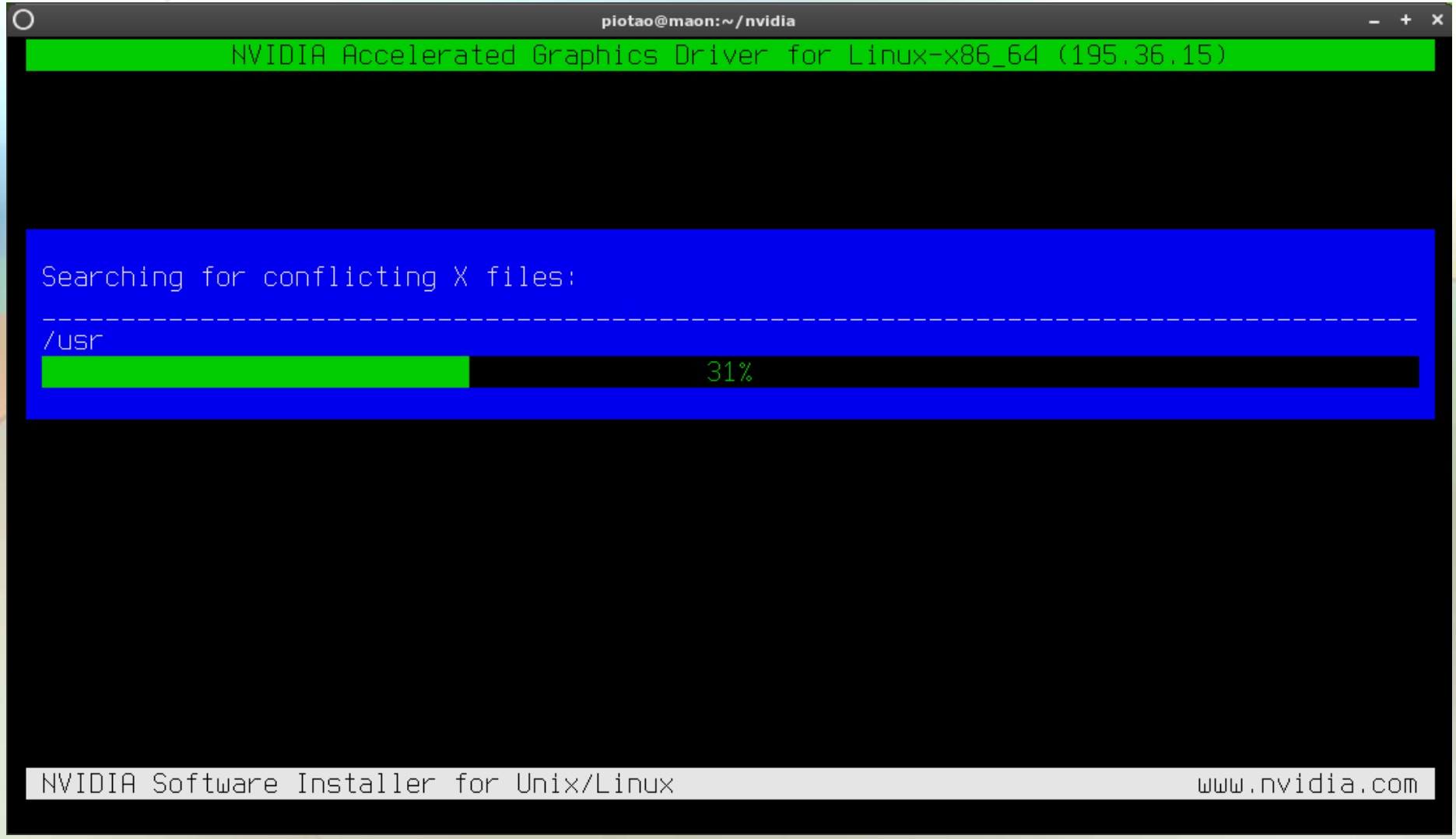
Return



Installation

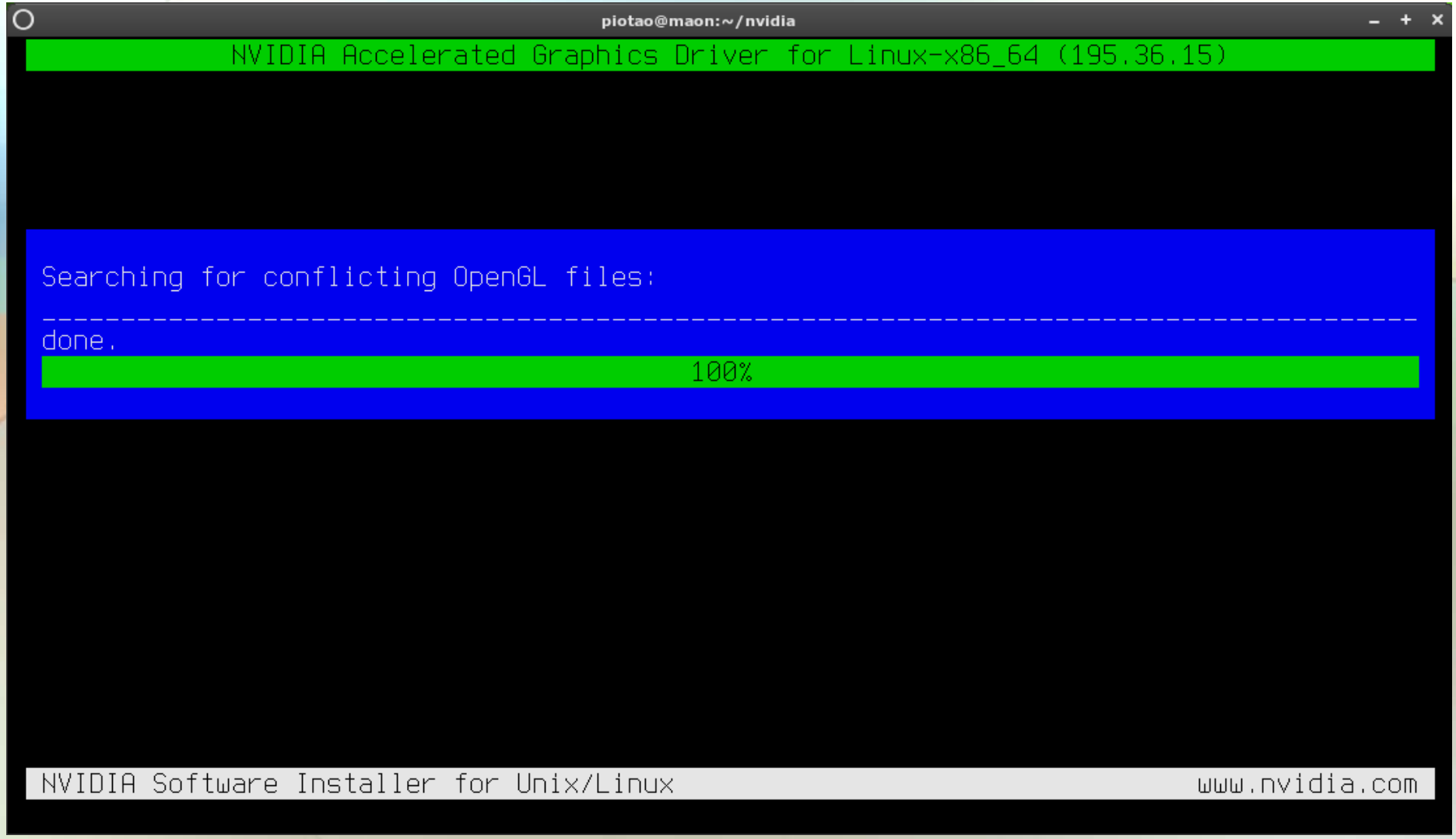
Linux CUDA installation

Return



Linux CUDA installation

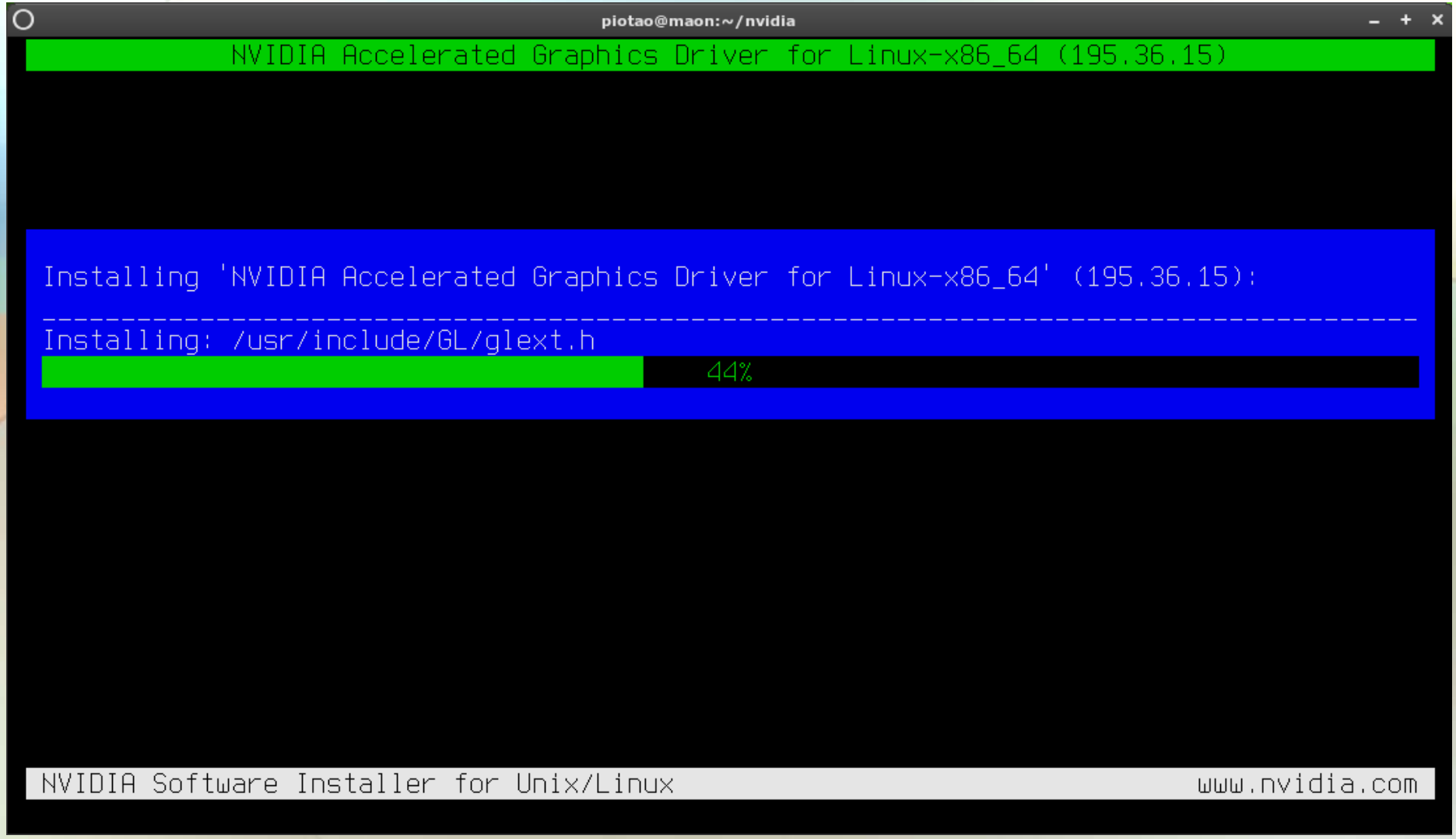
Return



Installation

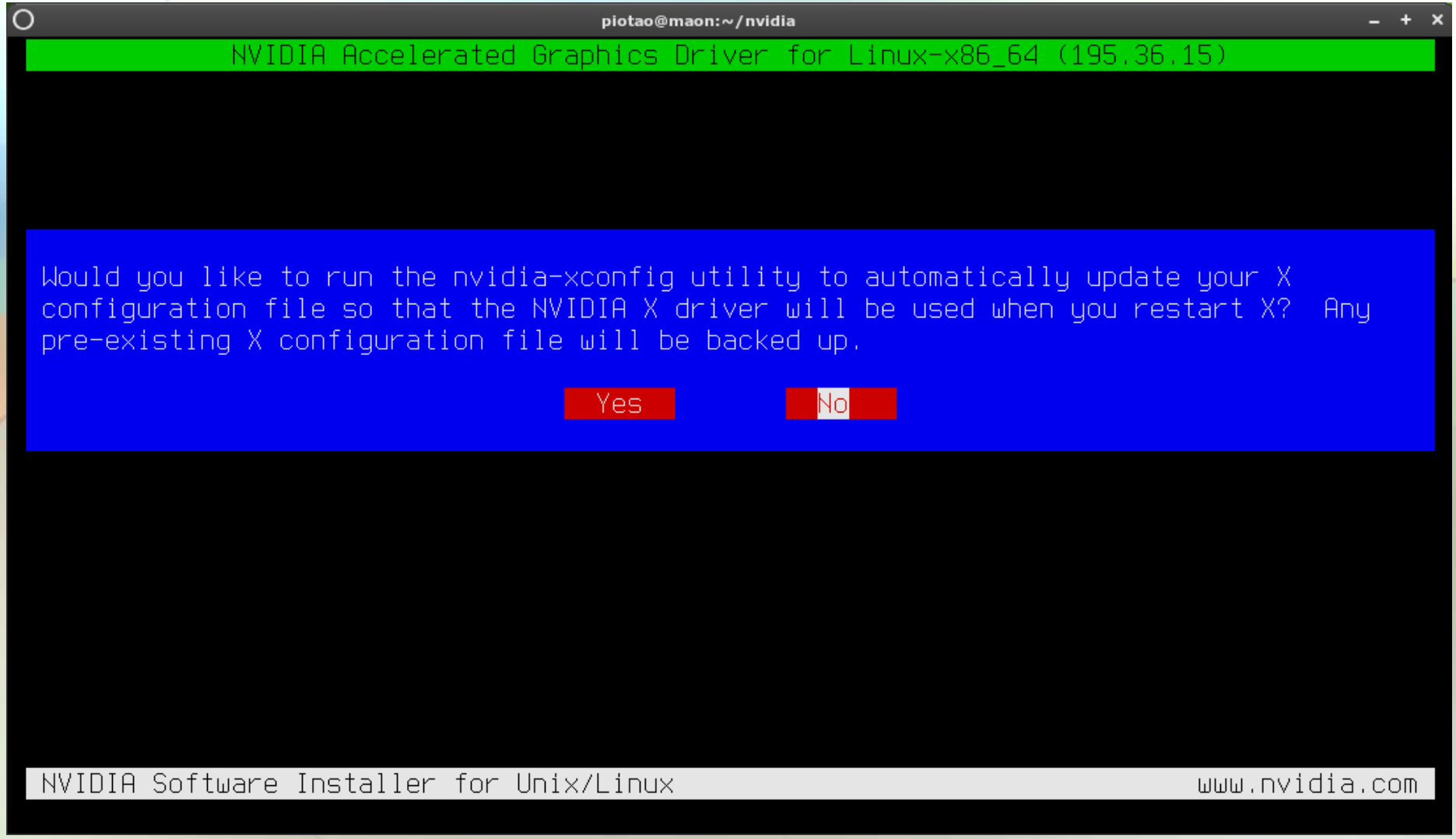
Linux CUDA installation

Return



Linux CUDA installation

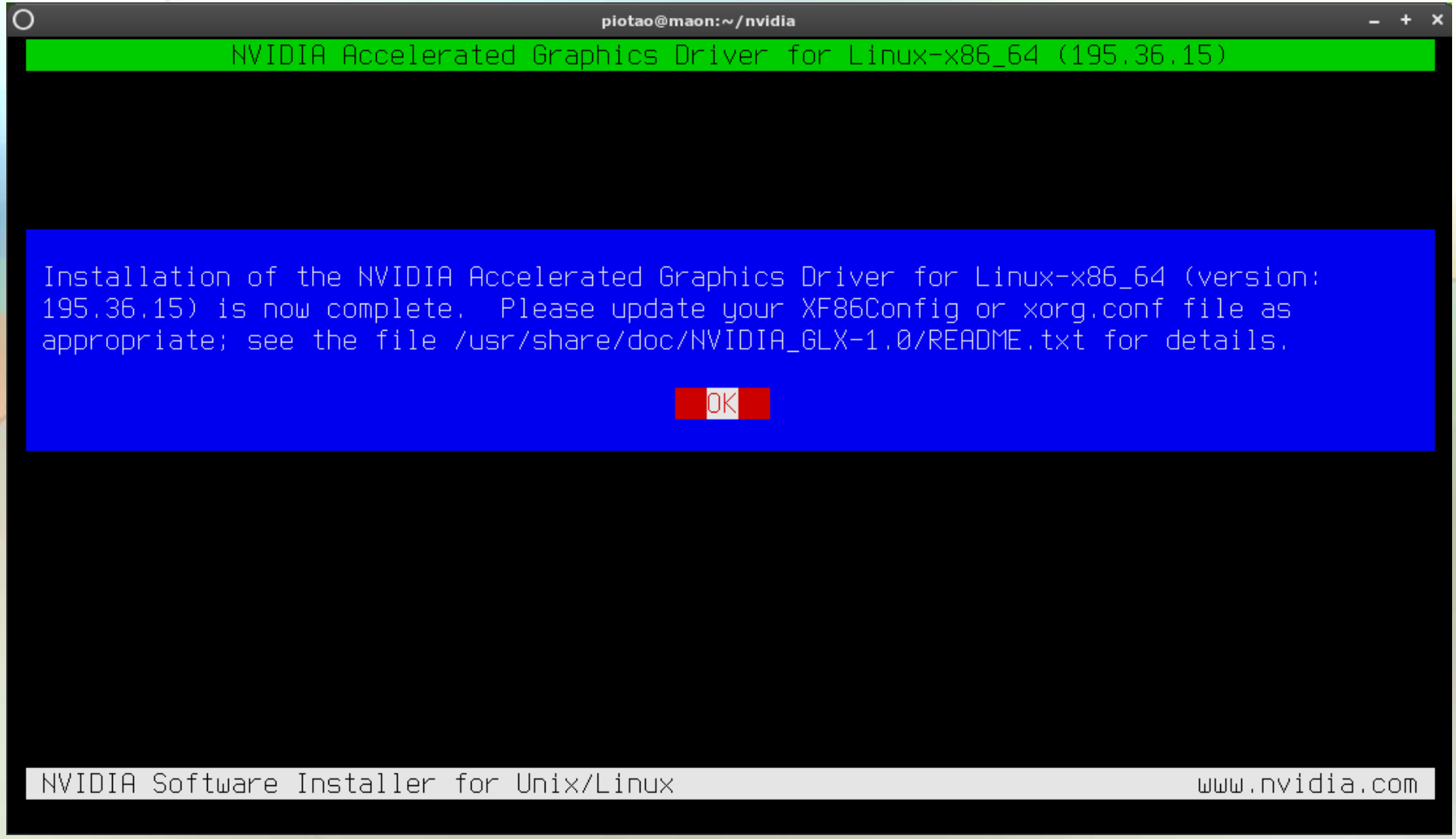
Return



Installation

Linux CUDA installation

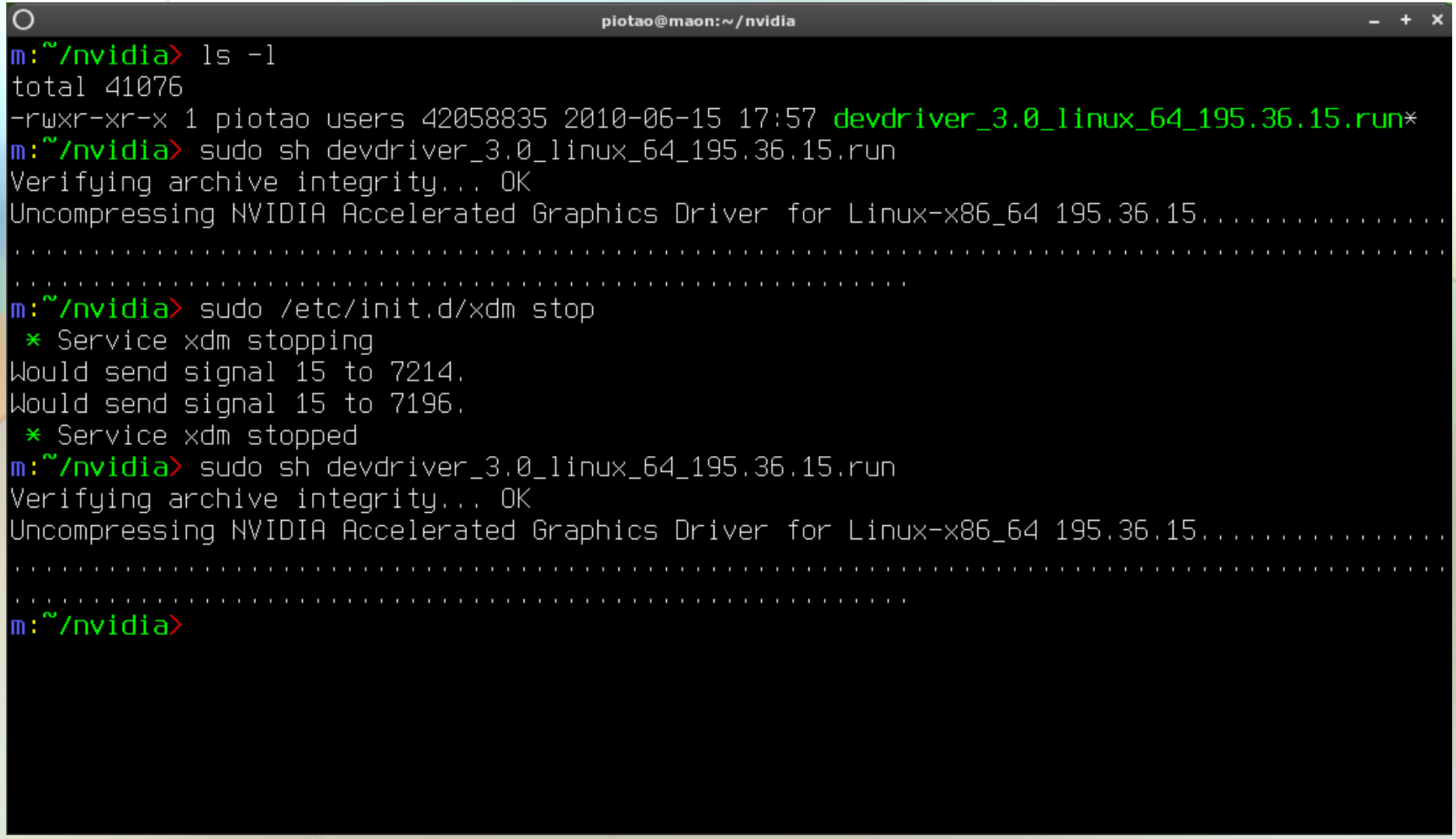
Return



Installation

Linux CUDA installation

Return



Installation

Linux CUDA installation

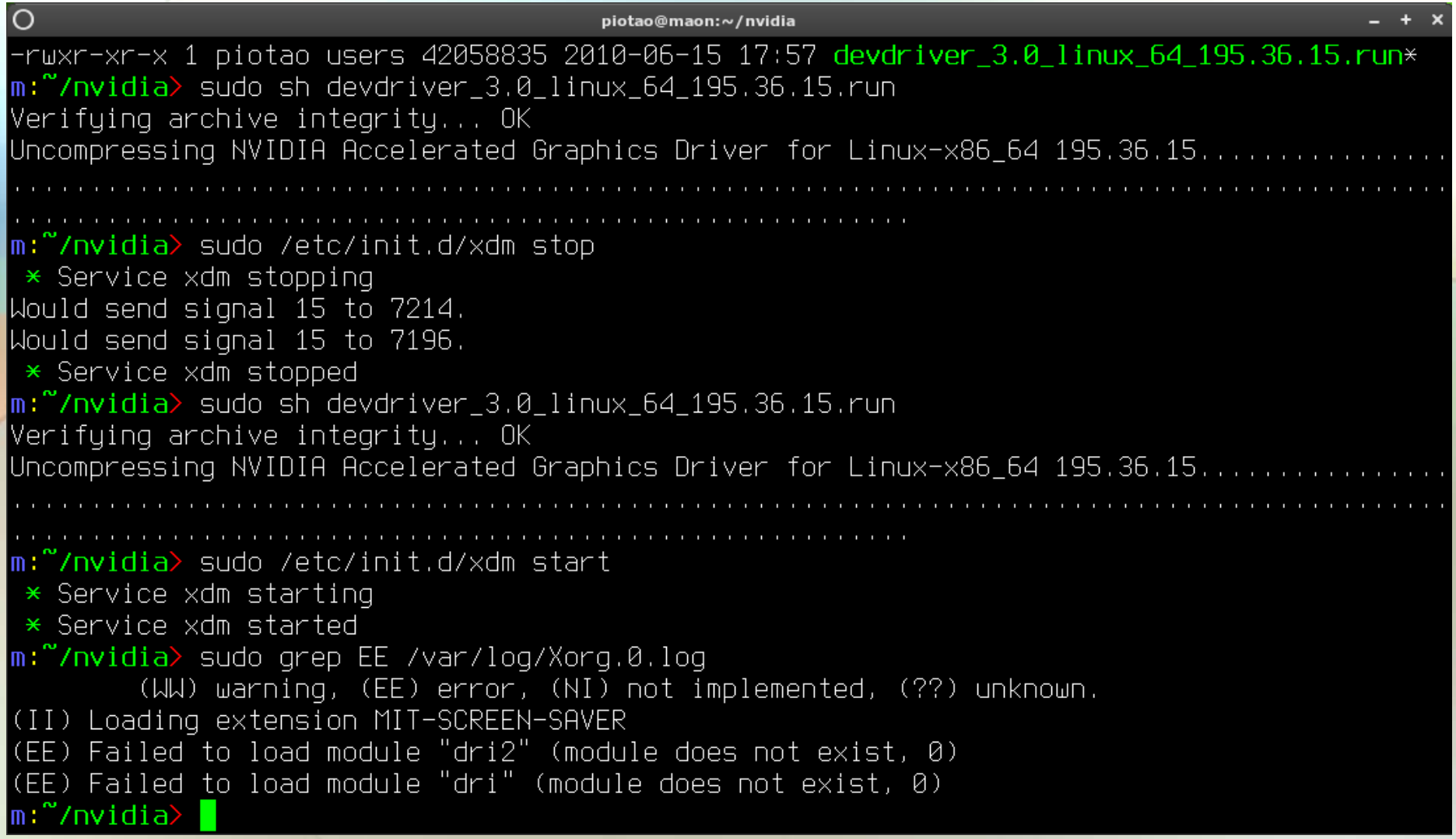
Return

```
piotao@maon:~/nvidia
m:~/nvidia> ls -l
total 41076
-rwxr-xr-x 1 piotao users 42058835 2010-06-15 17:57 devdriver_3.0_linux_64_195.36.15.run*
m:~/nvidia> sudo sh devdriver_3.0_linux_64_195.36.15.run
Verifying archive integrity... OK
Uncompressing NVIDIA Accelerated Graphics Driver for Linux-x86_64 195.36.15.............
.....................................................................................
.....................................................
m:~/nvidia> sudo /etc/init.d/xdm stop
 * Service xdm stopping
Would send signal 15 to 7214.
Would send signal 15 to 7196.
 * Service xdm stopped
m:~/nvidia> sudo sh devdriver_3.0_linux_64_195.36.15.run
Verifying archive integrity... OK
Uncompressing NVIDIA Accelerated Graphics Driver for Linux-x86_64 195.36.15.............
.....................................................................................
.....................................................
m:~/nvidia> sudo /etc/init.d/xdm start
 * Service xdm starting
 * Service xdm started
m:~/nvidia>
```

Installation

Linux CUDA installation

Return



Installation

Linux CUDA installation

Return to CUDA Linux Installation

CUDA Assembler example

`nvcc --ptx`

- Sample of the code:

piotao@venona:~/docs/Lectures/NVIDIA-CUDA2/code/gpu

```
        mov.u16         %rh2, %ctaid.x;         //
        mov.u16         %rh3, %ntid.y;          //
        mov.u16         %rh4, %ctaid.y;         //
        ld.param.s32    %r3, [__cudaparm__Z12MatMulKernel6MatrixS_S__A+0];
// id:41 __cudaparm__Z12MatMulKernel6MatrixS_S__A+0x0
        mov.u32         %r4, 0;                 //
        setp.le.s32     %p1, %r3, %r4;          //
        mov.f32         %f1, 0f00000000;        // 0
        @%p1 bra        $Lt_0_2306;             //
        mov.s32         %r5, %r3;               //
        mul.wide.u16    %r6, %rh3, %rh4;        //
        mul.wide.u16    %r7, %rh1, %rh2;        //
        add.u32         %r8, %r6, %r2;          //
        add.u32         %r9, %r7, %r1;          //
        mul.lo.s32      %r10, %r3, %r8;         //
        mov.s32         %r11, %r10;             //
        add.s32         %r12, %r10, %r3;        //
        ld.param.s32    %r13, [__cudaparm__Z12MatMulKernel6MatrixS_S__B+0];
// id:43 __cudaparm__Z12MatMulKernel6MatrixS_S__B+0x0
        cvt.s64.s32     %rd1, %r13;             //
        mul.lo.u64      %rd2, %rd1, 4;          //
matrix.ptx lines 72-90/131 67%
```

Software

CUDA Assembler example

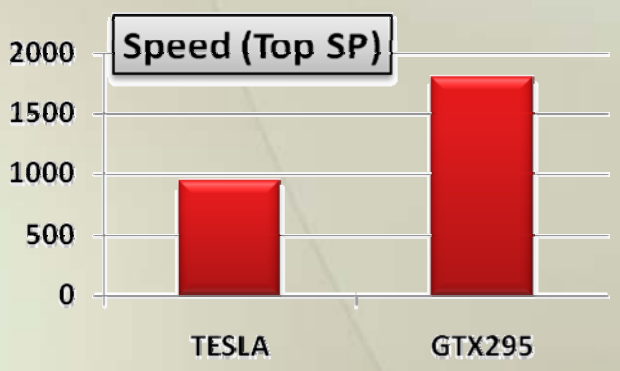
Return to CUDA Tools

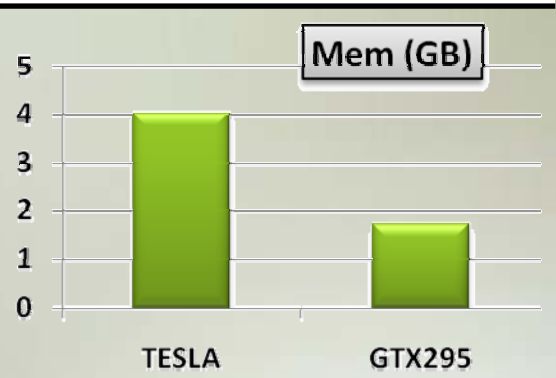
Hardware

Hardware view

NVIDIA TESLA 1060	
Processor Cores	240
Processor Clock	1.3 GHz
Single Precision floating point performance (peak)	933
Double precision floating point performance (peak)	78
Memory	4 GB GDDR3
Memory Clock	800 MHz
Memory Bandwidth	102 GB/sec
Max Power Consumption	187.8 W

NVIDIA GTX 295	
Processor Cores	480 (240 per GPU)
Procesor/Graphics Clock	1242 / 576 MHz
Single Precision floating point performance (peak)	1788
Double precision floating point performance (peak)	149
Memory	2 x 896 = 1792 MB
Memory Clock	999 MHz
Memory Bandwidth	223.8 GB/sec
Max Power Consumption	289 W (680 W System)

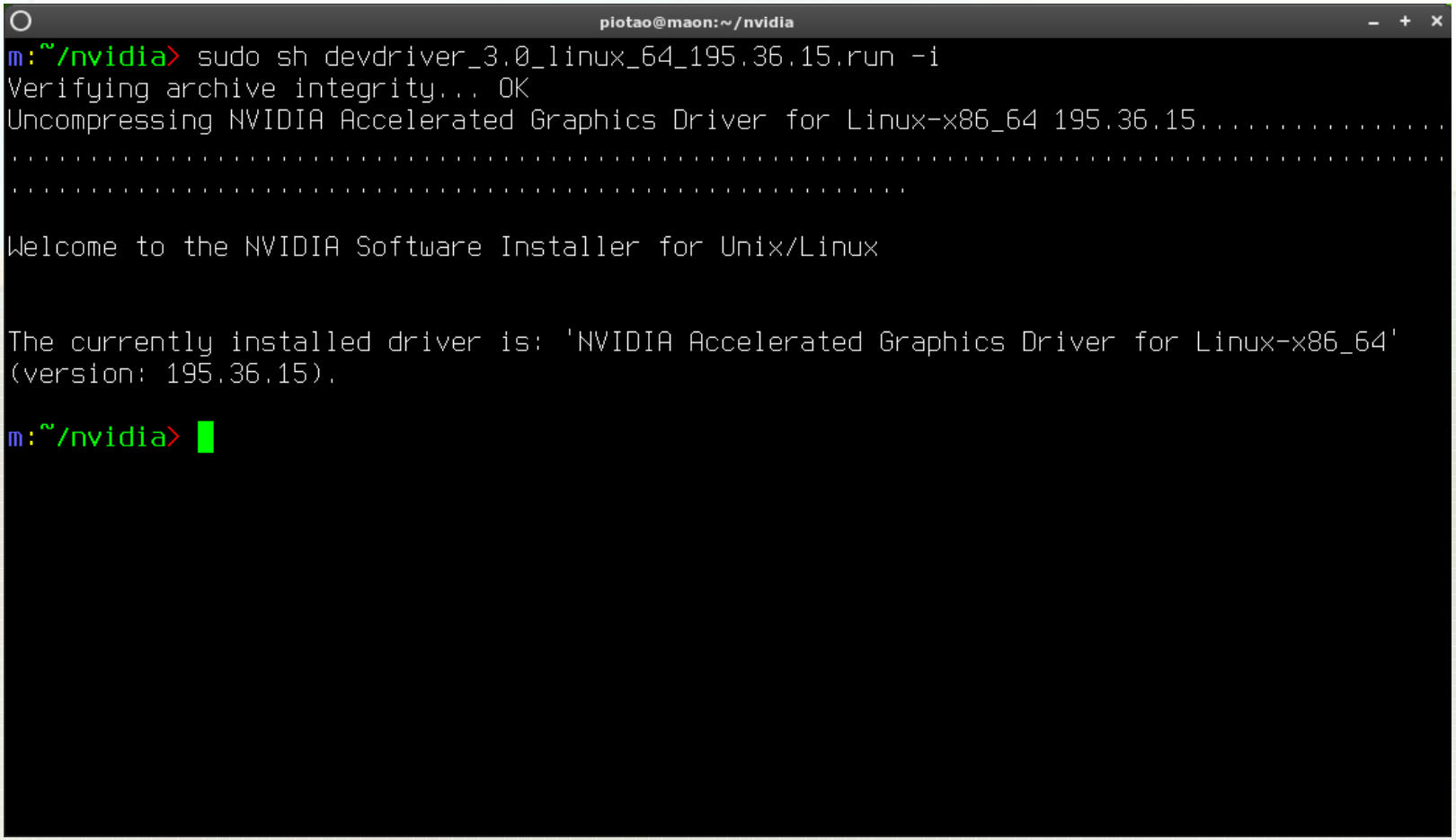




Installation

Verify installation under Linux

- Checking the CUDA SDK installation



Look why the programming can be hybrid

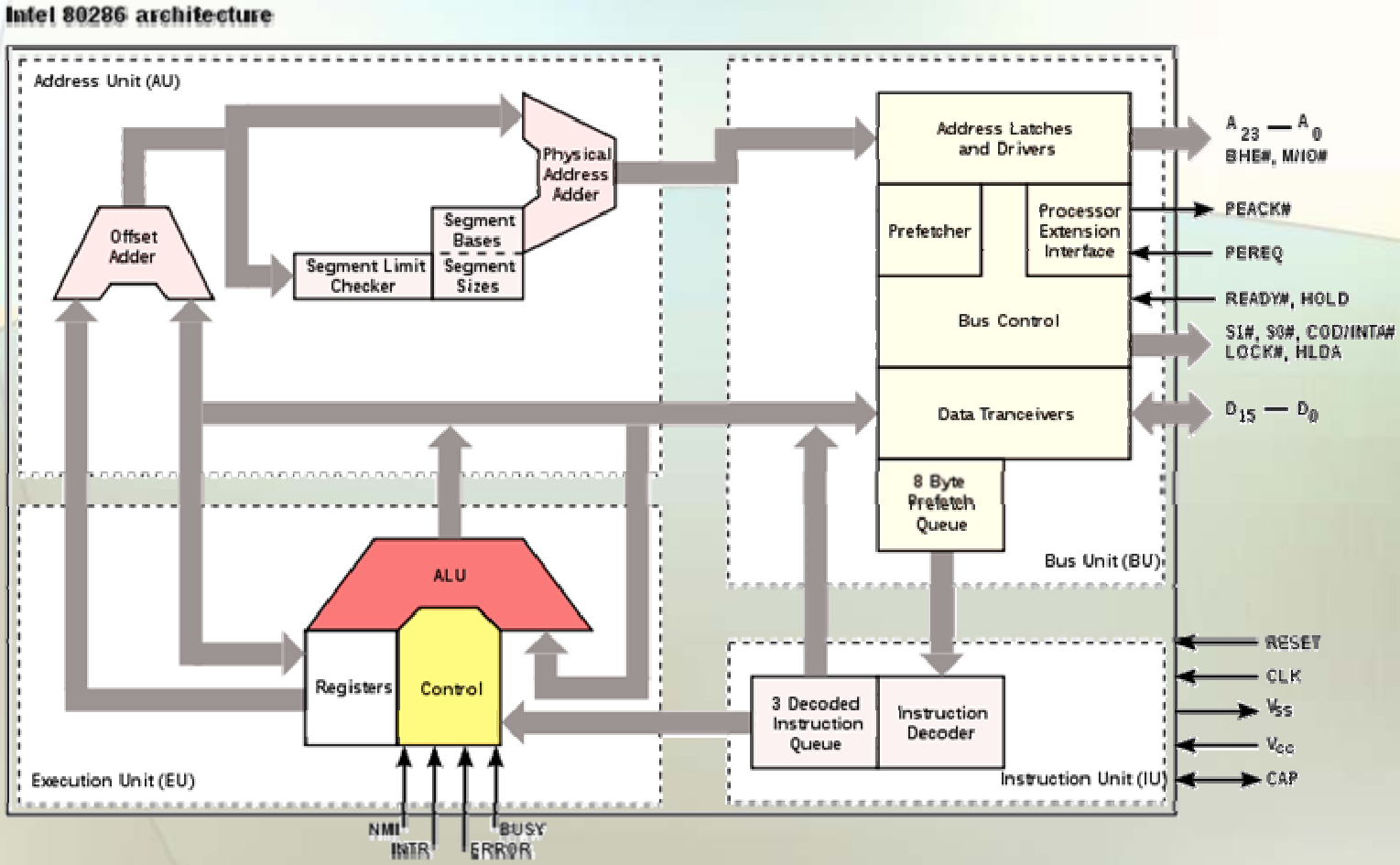
80286 and Cell X9i examples

The constant changes of paradigms

The reasons for hybrid programming

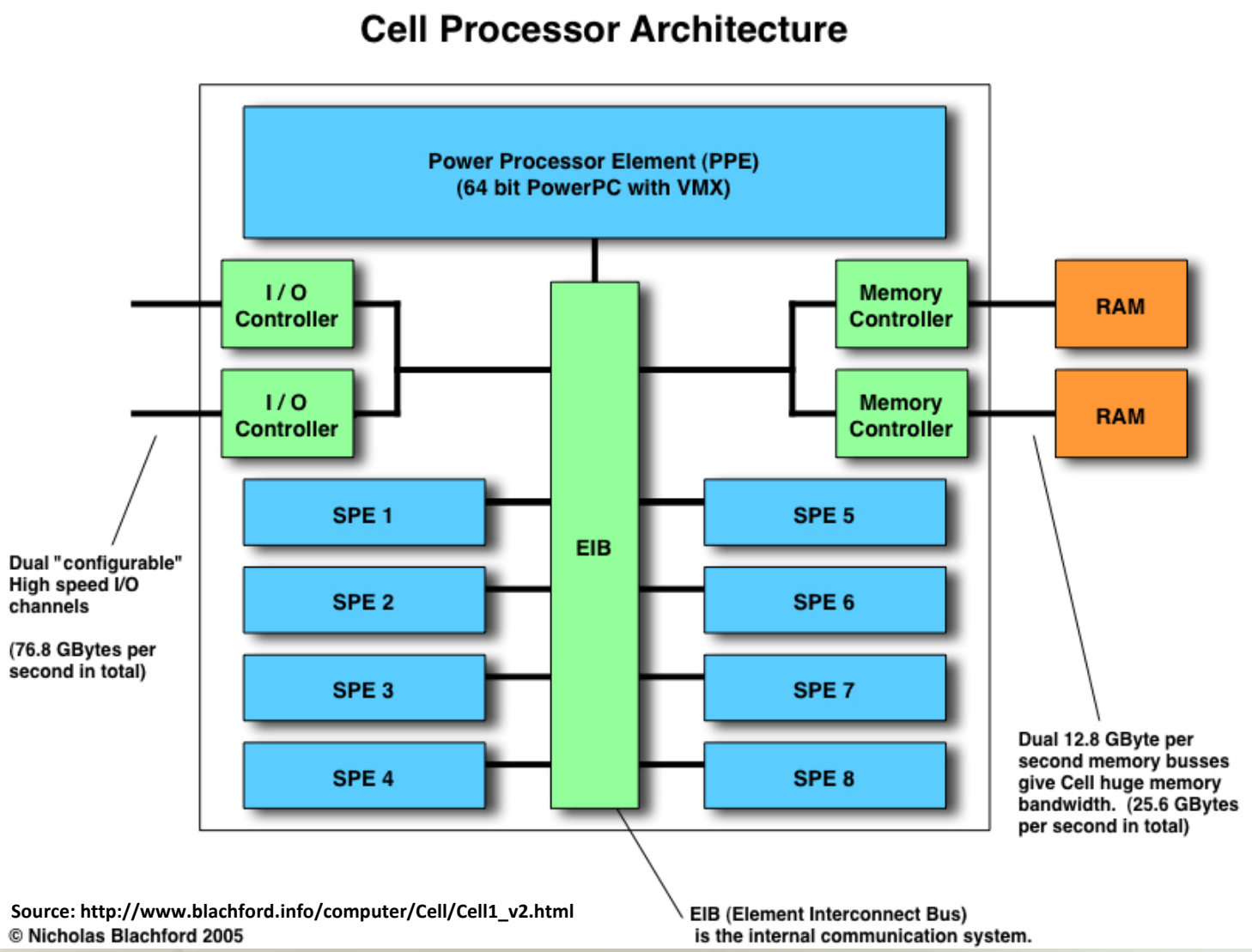
INTRODUCTION

Intel 80286 architecture



Creative Commons Attribution-Share Alike 3.0 Unported

Cell architecture



Source: http://www.blachford.info/computer/Cell/Cell1_v2.html

© Nicholas Blachford 2005

The Constant Change

- Paradigms change over the years.

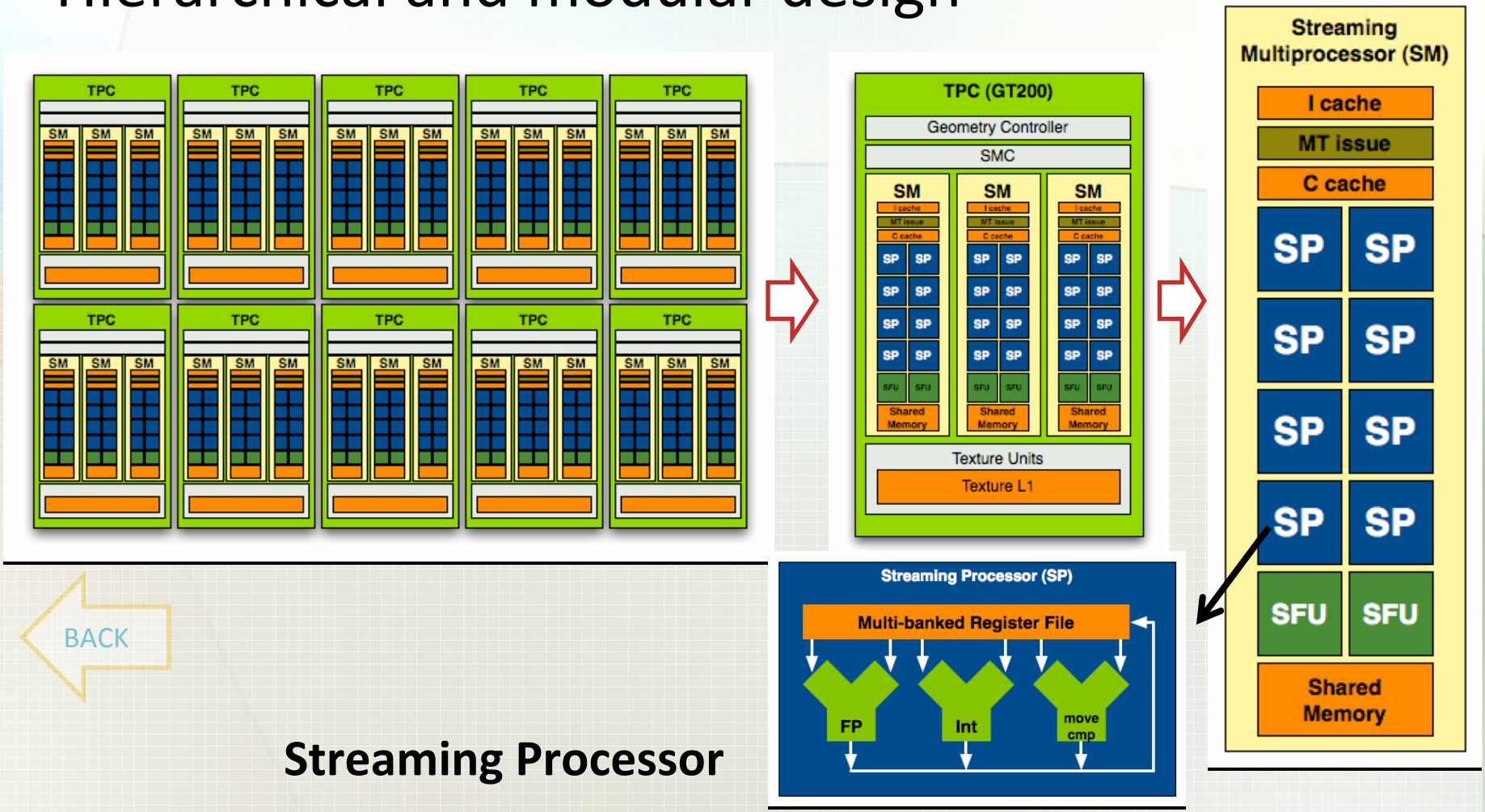
Yesterday	Today
• Power is cheap.	• Power is expansive.
• Transistors are expansive.	• Transistors are cheap.
• Multiplication is slow.	• Multiplication is fast.
• Load/store is fast.	• Load/store is slow.
• Complexity level can be vastly increased	• Complexity level can't be increased (physical barriers)
• ILP is possible and effective	• ILP is not effective
• Increasing clock frequency	• Increasing parallelism
• Any speedup<1x = FAILURE	• Any speedup = SUCCESS

The reasons of „hybrid” - summary

- Numeric calculations can be performed on hybrid architectures efficiently.
- Many numerical algorithms have the hybrid structure. The most computationally intensive components can be accelerated.
- Paradigms change and we have to evade physical limitations.
- Hybrid technologies are cost-effective and power-efficient.

Hardware view

- Hierarchical and modular design



Streaming Processor