CUDA

Compute Unified Device Architecture

ou

Arquitetura de Dispositivo de Computação Unificada

Nvidia Corporation

- É uma API destinada a computação paralela, GPGPU, e computação heterogênea, criada pela Nvidia, destinada a placas gráficas que suportem a API (normalmente placas gráficas com chipset da Nvidia).

- A plataforma CUDA dá acesso ao conjunto de instruções da GPU e a elementos de computação paralela, para a execução de núcleos de computação.

O QUE É CUDA

- No final dos anos 90, surgiu a primeira GPU da NVIDIA, quando o hardware começou a tornar-se cada vez mais programável.

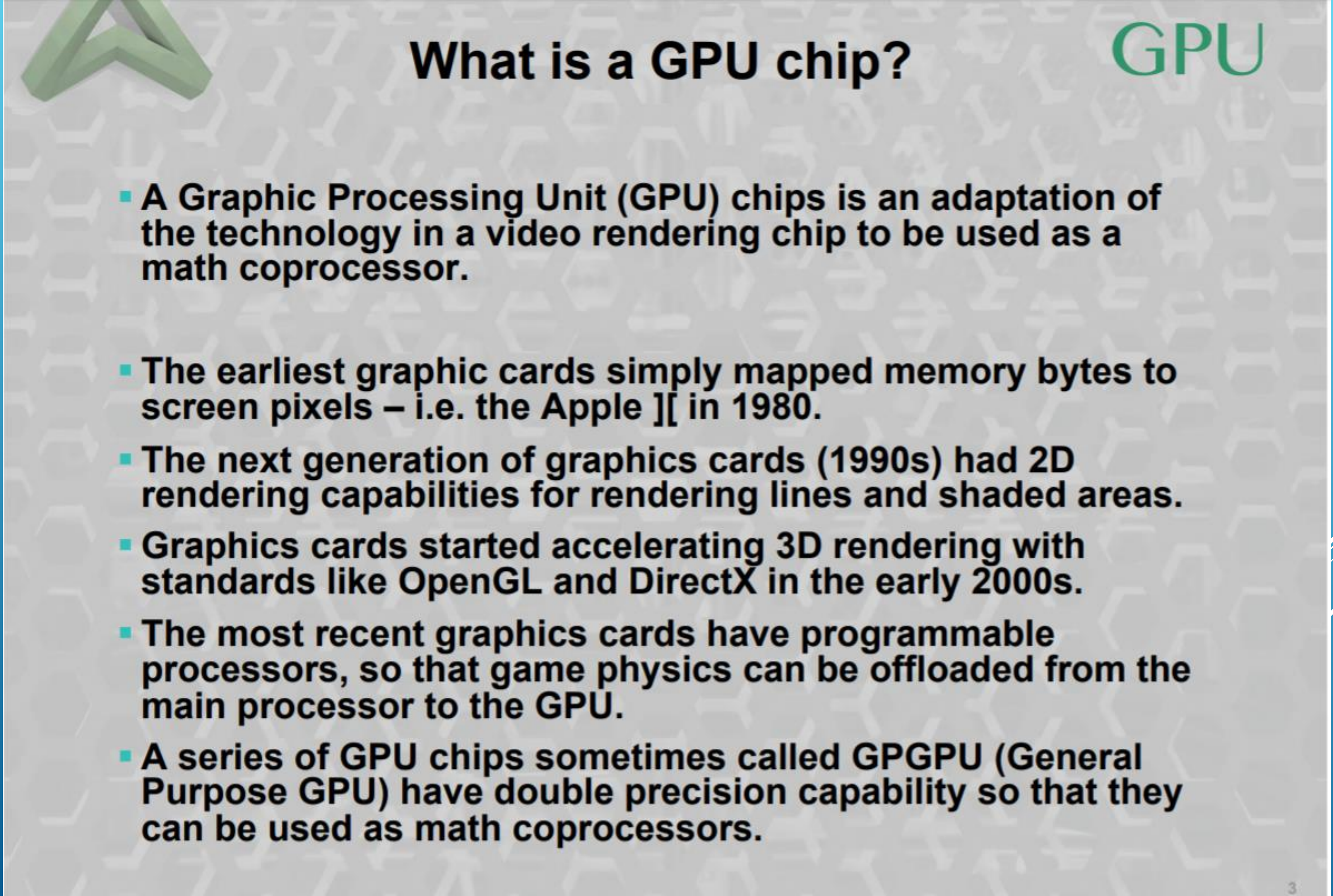
- Em 2003, um grupo de pesquisadores liderado por Ian Buck desenvolveu o primeiro modelo de programação a adotar a linguagem C em uma plataforma de computação paralela, revelando assim uma GPU, como um processador de propósito geral em uma linguagem de alto nível, além de os programas serem sete vezes mais rápidos.

HISTÓRIA

- A NVIDIA então, investiu num hardware extremamente rápido e convidou Ian Buck para trabalhar na empresa e começar a desenvolver uma solução para executar o C na GPU de forma melhor.

- Assim, a NVIDIA apresentou em 2006 o CUDA, a primeira solução para computação de propósito geral em GPUs.

HISTÓRIA

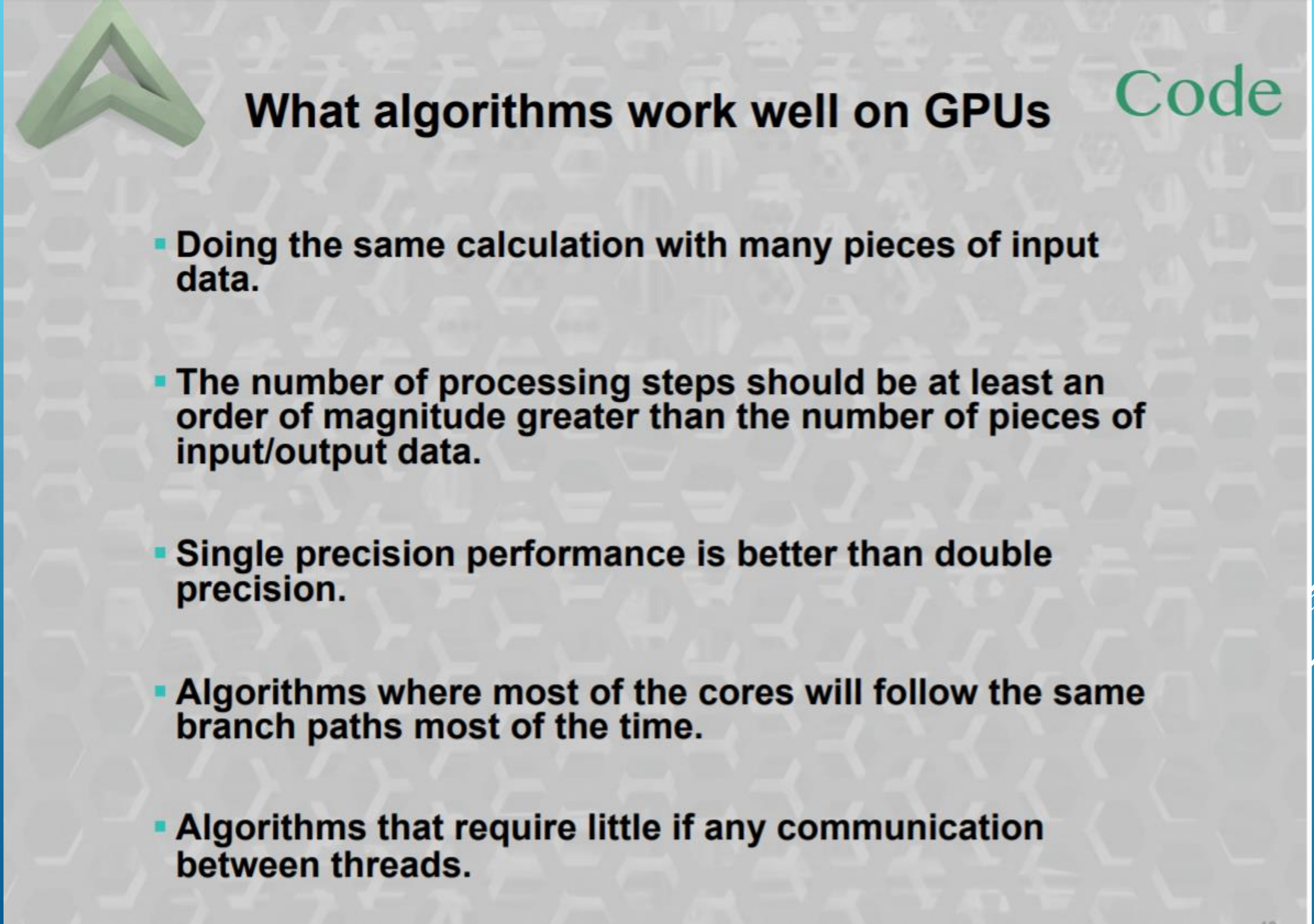


What is a GPU chip?

GPU

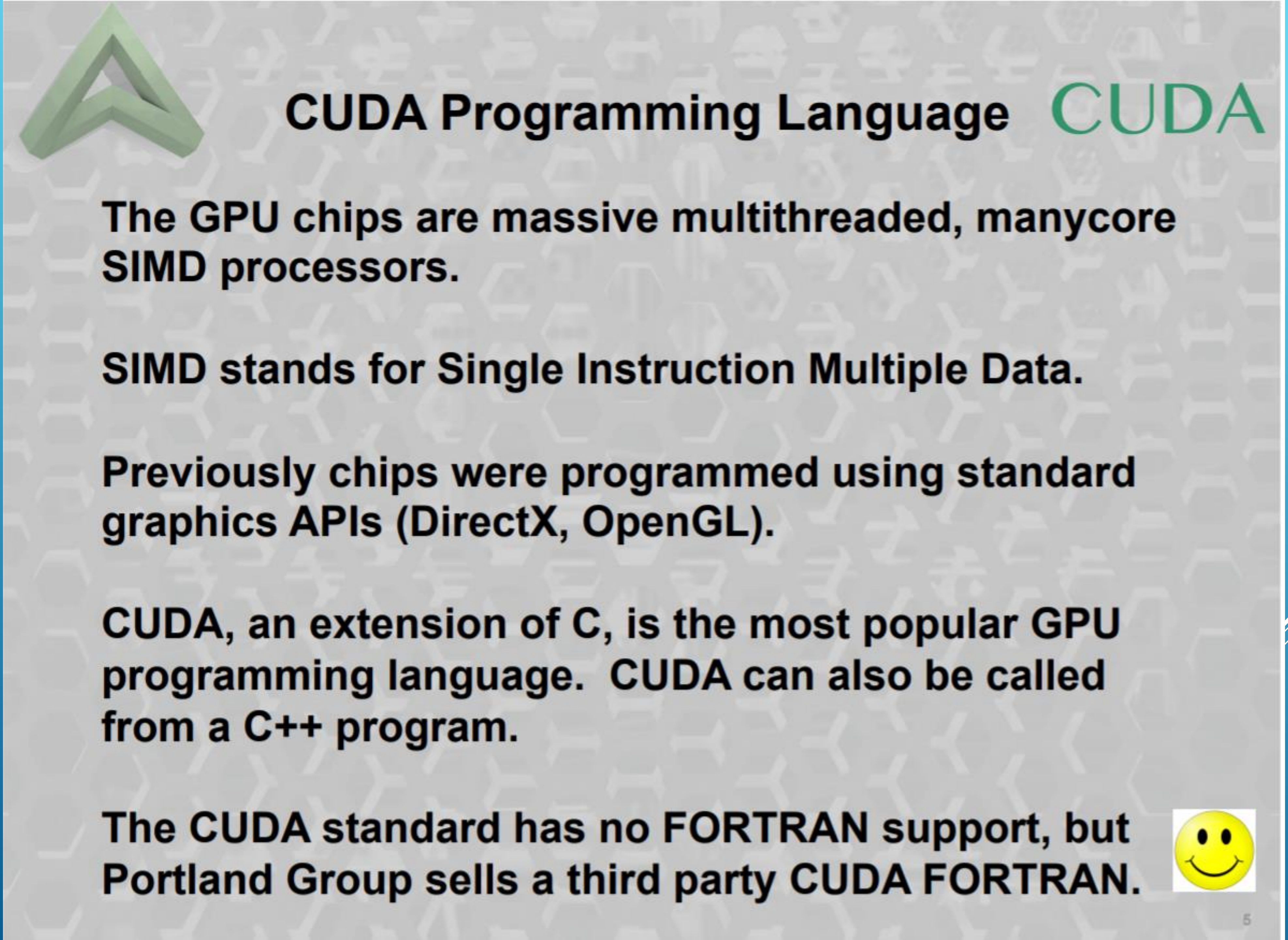
- A Graphic Processing Unit (GPU) chips is an adaptation of the technology in a video rendering chip to be used as a math coprocessor.

- The earliest graphic cards simply mapped memory bytes to screen pixels – i.e. the Apple][in 1980.
- The next generation of graphics cards (1990s) had 2D rendering capabilities for rendering lines and shaded areas.
- Graphics cards started accelerating 3D rendering with standards like OpenGL and DirectX in the early 2000s.
- The most recent graphics cards have programmable processors, so that game physics can be offloaded from the main processor to the GPU.
- A series of GPU chips sometimes called GPGPU (General Purpose GPU) have double precision capability so that they can be used as math coprocessors.



What algorithms work well on GPUs

- Doing the same calculation with many pieces of input data.
- The number of processing steps should be at least an order of magnitude greater than the number of pieces of input/output data.
- Single precision performance is better than double precision.
- Algorithms where most of the cores will follow the same branch paths most of the time.
- Algorithms that require little if any communication between threads.



CUDA Programming Language

CUDA

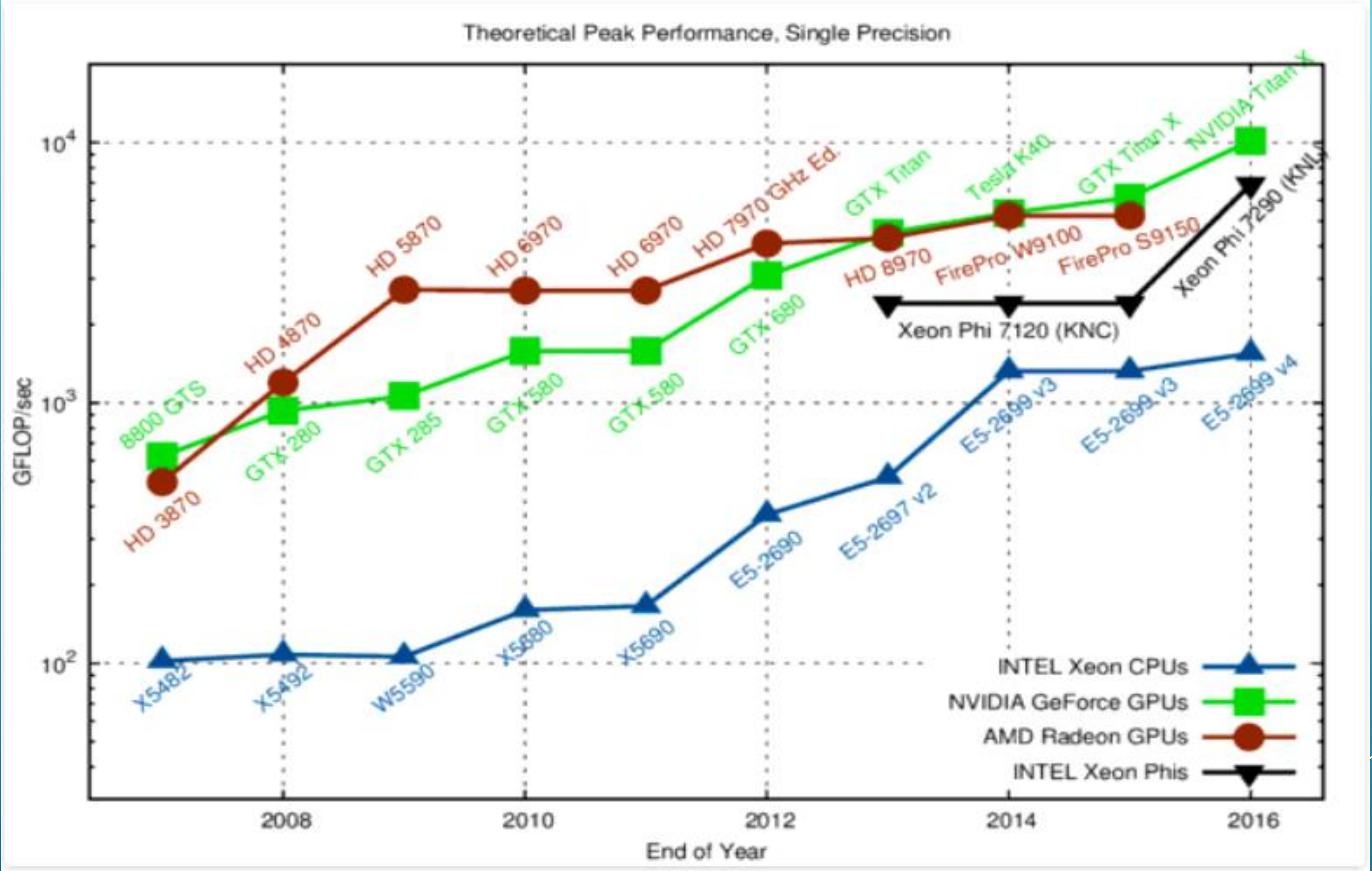
The GPU chips are massive multithreaded, manycore SIMD processors.

SIMD stands for Single Instruction Multiple Data.

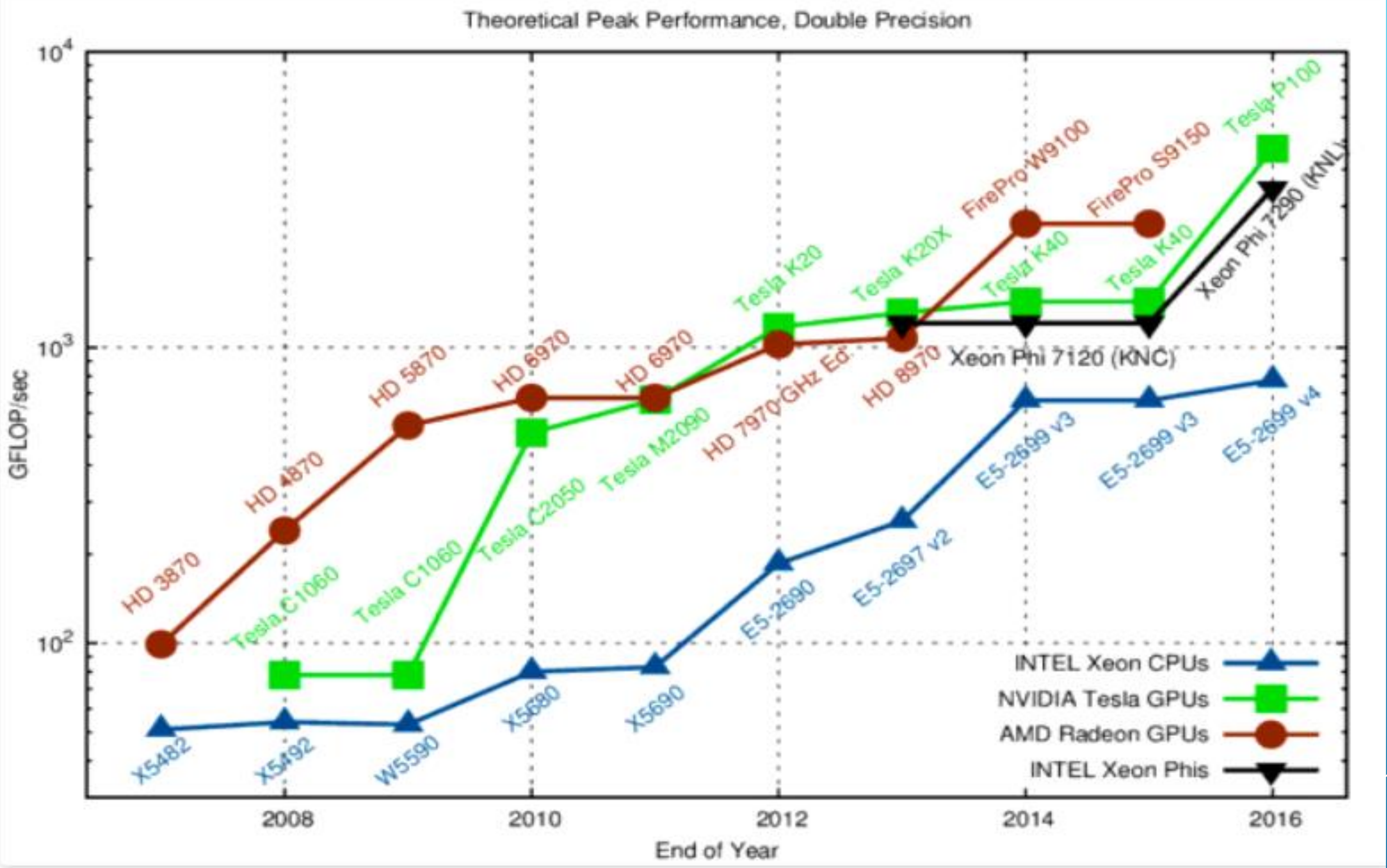
Previously chips were programmed using standard graphics APIs (DirectX, OpenGL).

CUDA, an extension of C, is the most popular GPU programming language. CUDA can also be called from a C++ program.

The CUDA standard has no FORTRAN support, but Portland Group sells a third party CUDA FORTRAN.

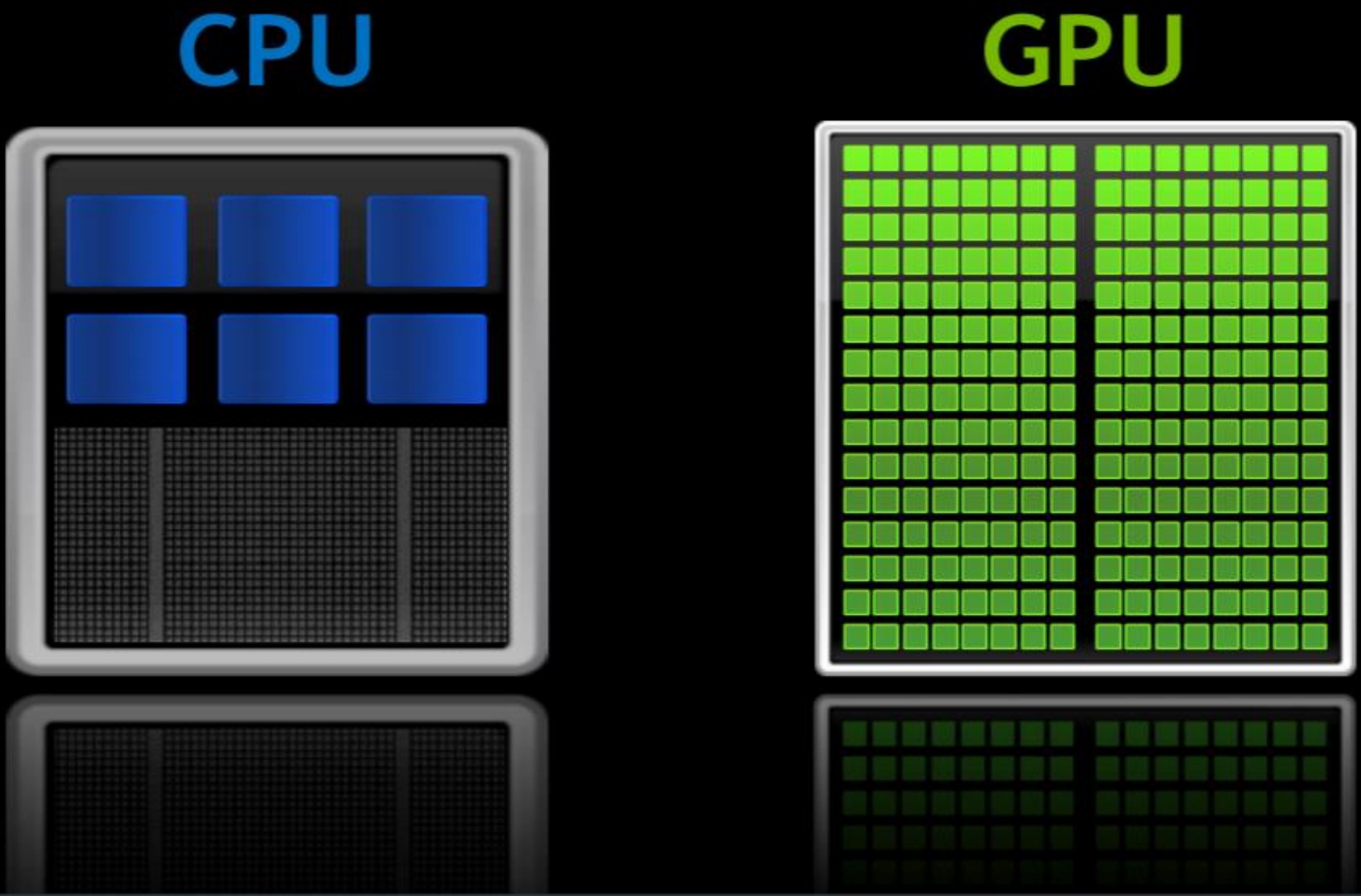


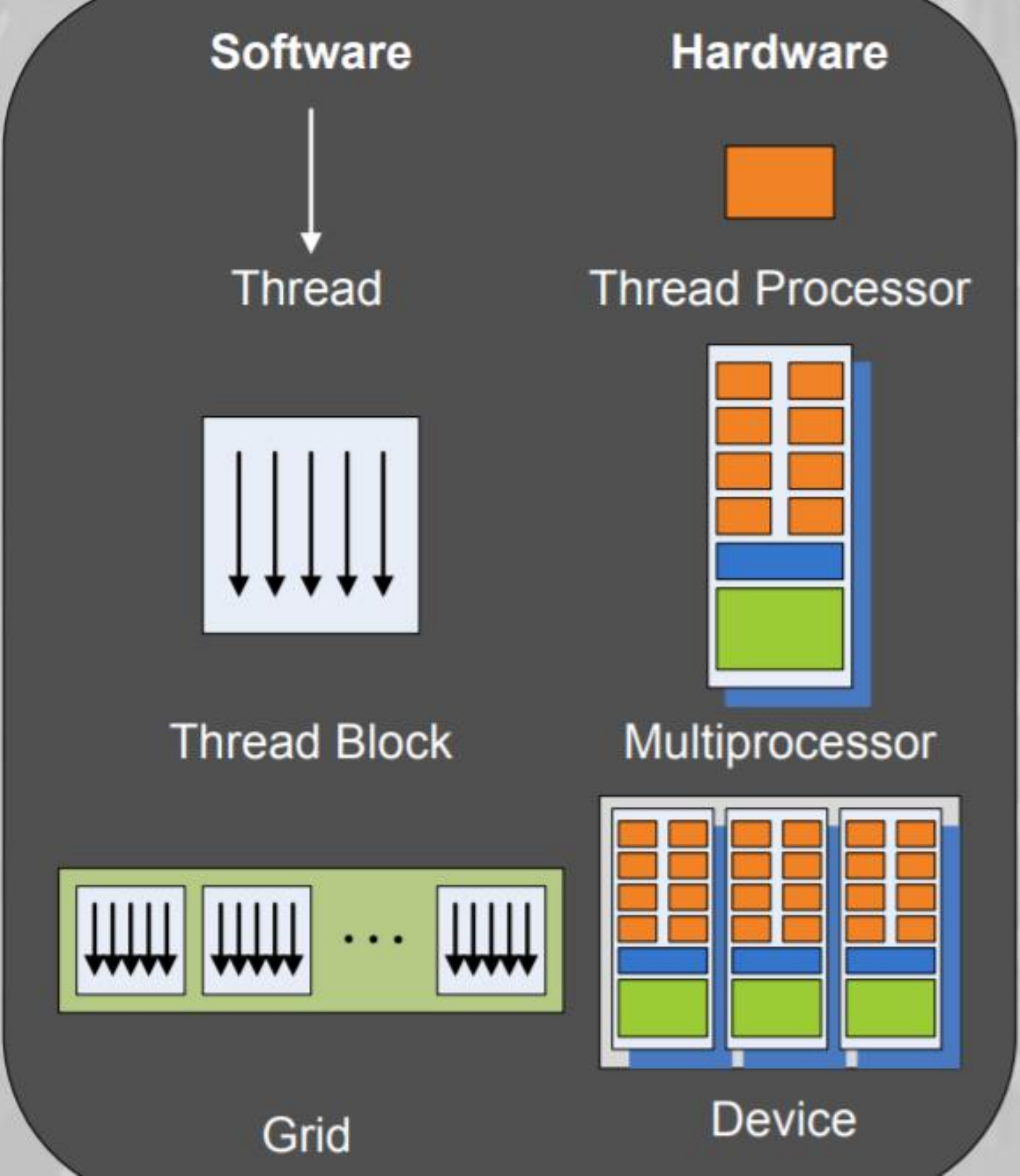
Comparison of theoretical peak GFLOP/sec in single precision. CPU data is for a single socket. Higher is better.



Comparison of theoretical peak GFLOP/sec in double precision. CPU data is for a single socket. Higher is better.

Heterogeneous Computing

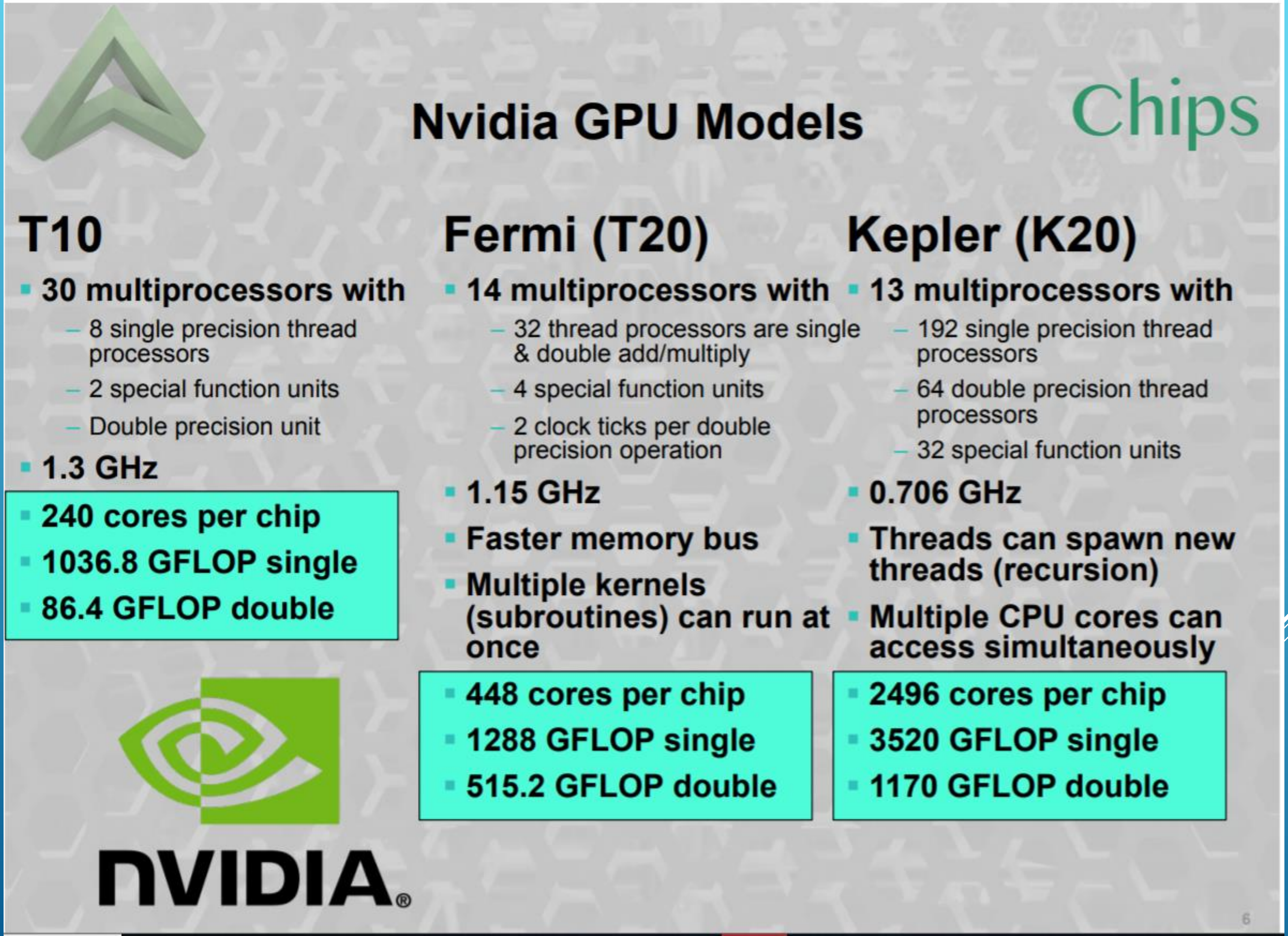




Thread is a single execution of a kernel, and all execute the same code

Threads within a block have access to shared memory for local cooperation

Kernel launched as a grid of independent thread blocks, and only a single kernel executes at a time (on T10)



Chips

Nvidia GPU Models

T10

- 30 multiprocessors with
  - 8 single precision thread processors
  - 2 special function units
  - Double precision unit
- 1.3 GHz
- 240 cores per chip
- 1036.8 GFLOP single
- 86.4 GFLOP double

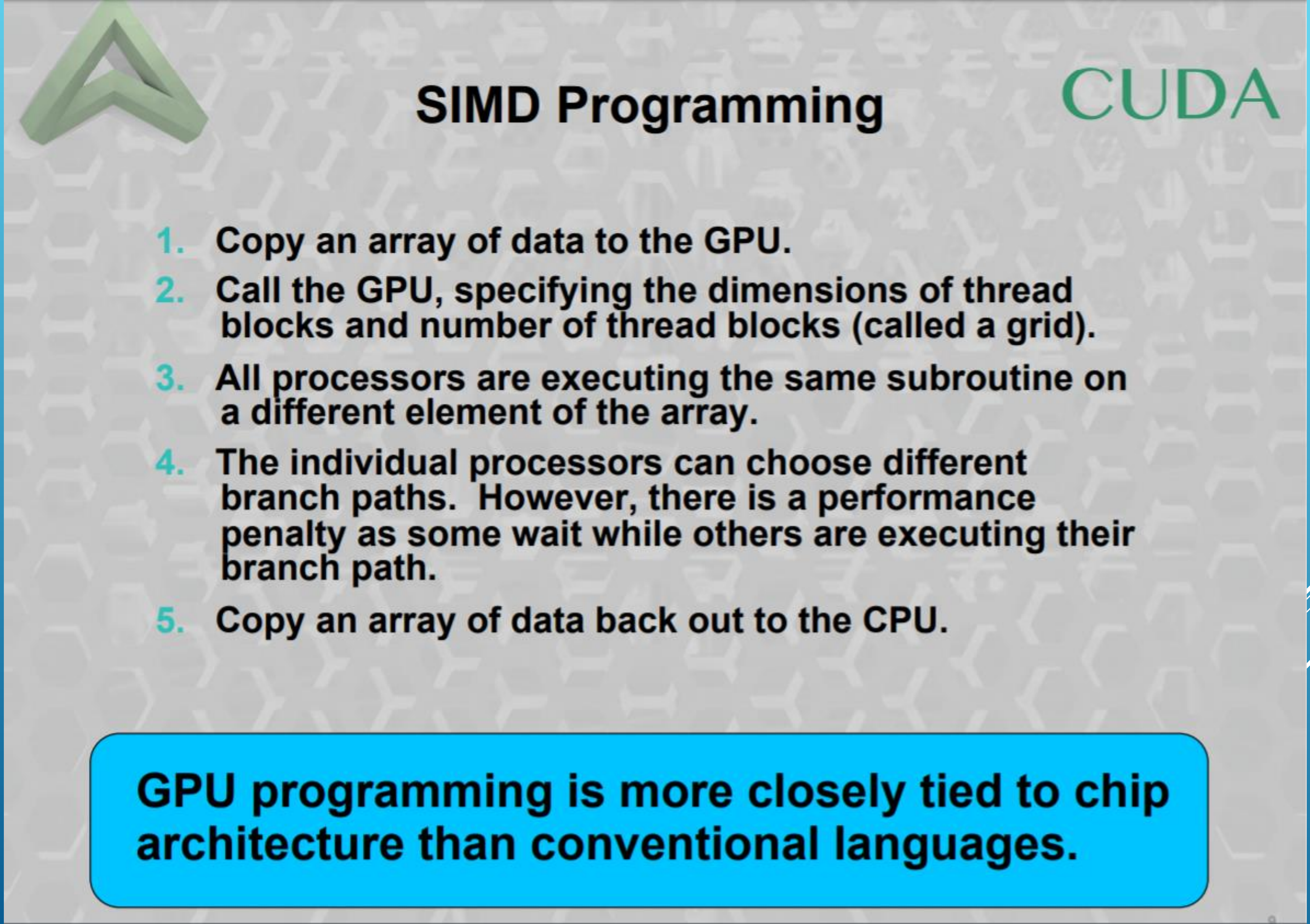
Fermi (T20)

- 14 multiprocessors with
  - 32 thread processors are single & double add/multiply
  - 4 special function units
  - 2 clock ticks per double precision operation
- 1.15 GHz
- Faster memory bus
- Multiple kernels (subroutines) can run at once
- 448 cores per chip
- 1288 GFLOP single
- 515.2 GFLOP double

Kepler (K20)

- 13 multiprocessors with
  - 192 single precision thread processors
  - 64 double precision thread processors
  - 32 special function units
- 0.706 GHz
- Threads can spawn new threads (recursion)
- Multiple CPU cores can access simultaneously
- 2496 cores per chip
- 3520 GFLOP single
- 1170 GFLOP double

NVIDIA®

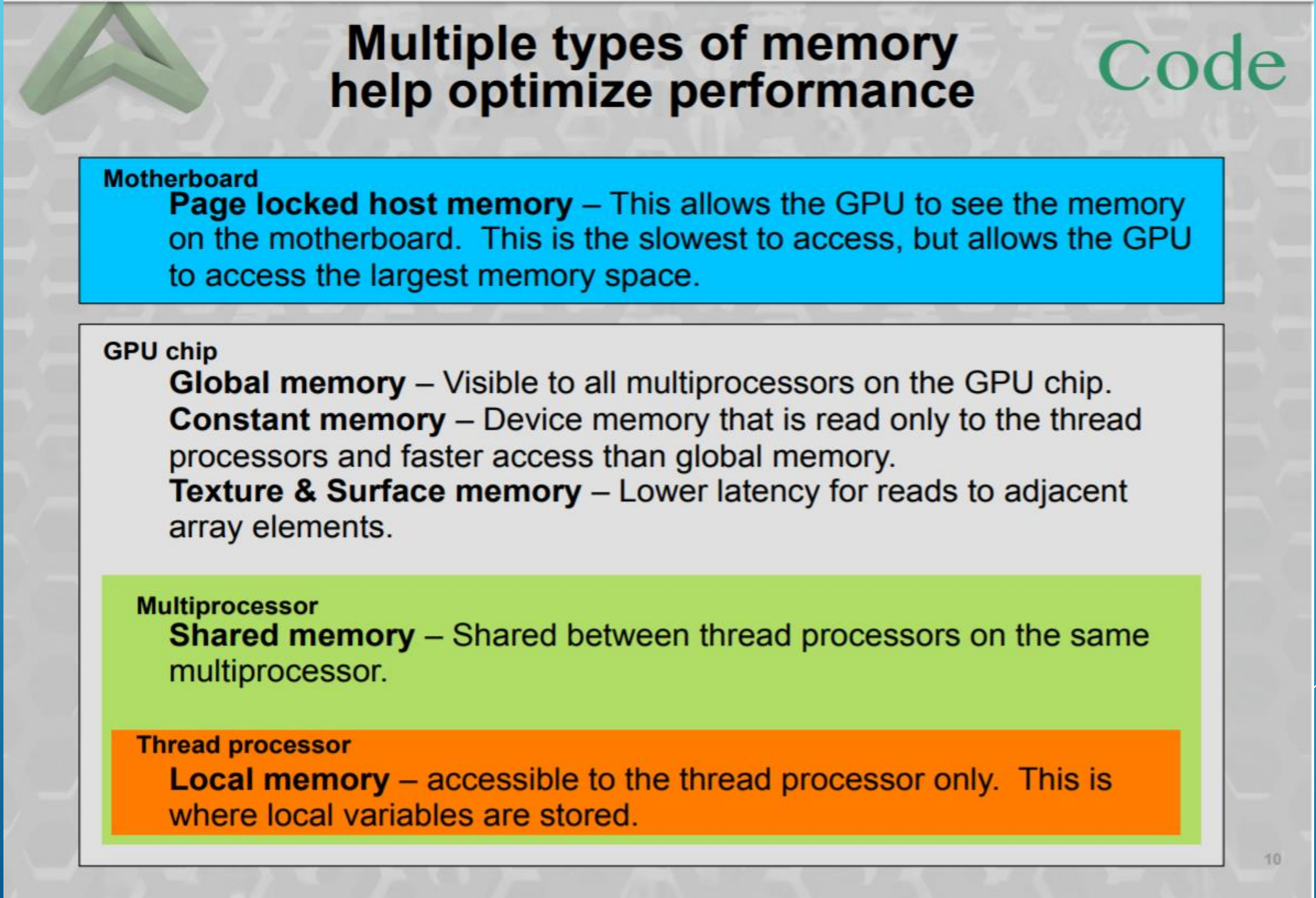


SIMD Programming

CUDA

1. Copy an array of data to the GPU.
2. Call the GPU, specifying the dimensions of thread blocks and number of thread blocks (called a grid).
3. All processors are executing the same subroutine on a different element of the array.
4. The individual processors can choose different branch paths. However, there is a performance penalty as some wait while others are executing their branch path.
5. Copy an array of data back out to the CPU.

GPU programming is more closely tied to chip architecture than conventional languages.



Multiple types of memory help optimize performance

Motherboard

Page locked host memory – This allows the GPU to see the memory on the motherboard. This is the slowest to access, but allows the GPU to access the largest memory space.

GPU chip

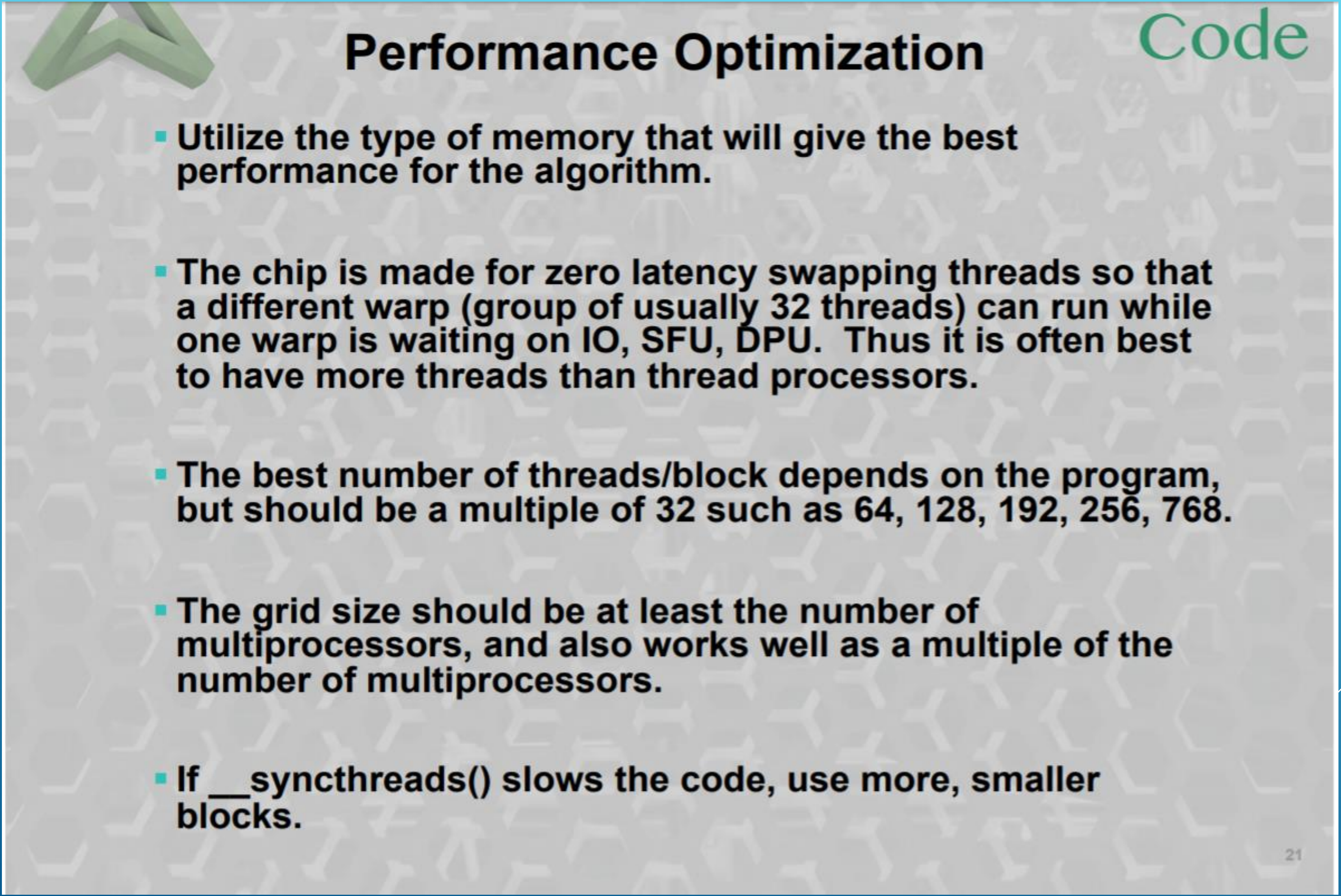
Global memory – Visible to all multiprocessors on the GPU chip.
Constant memory – Device memory that is read only to the thread processors and faster access than global memory.
Texture & Surface memory – Lower latency for reads to adjacent array elements.

Multiprocessor

Shared memory – Shared between thread processors on the same multiprocessor.

Thread processor

Local memory – accessible to the thread processor only. This is where local variables are stored.

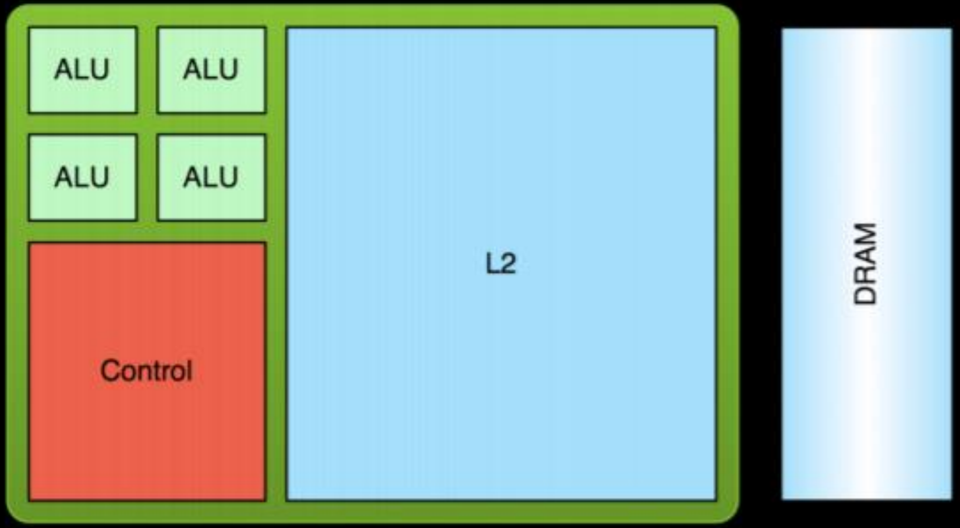


Performance Optimization

Code

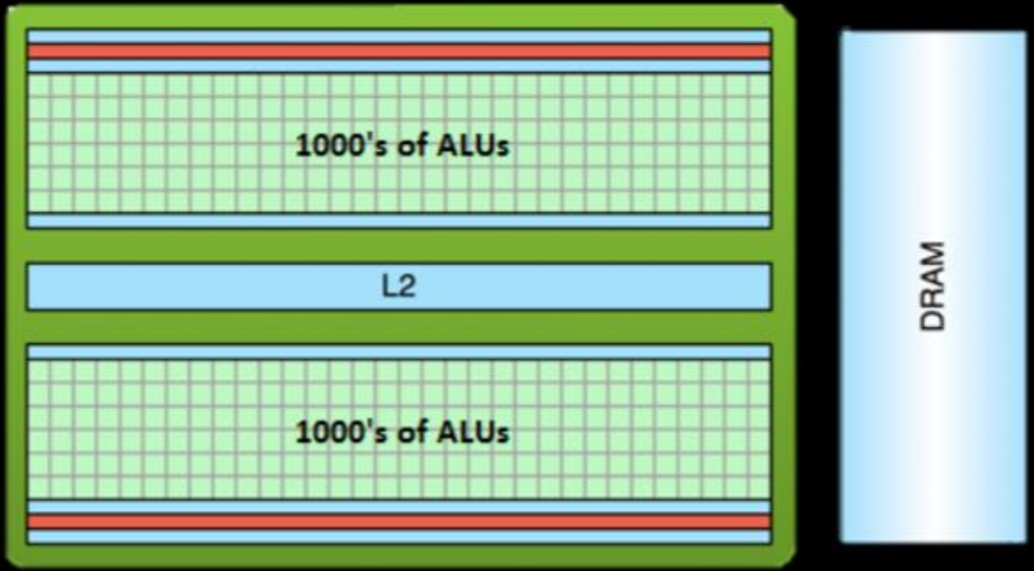
- Utilize the type of memory that will give the best performance for the algorithm.
- The chip is made for zero latency swapping threads so that a different warp (group of usually 32 threads) can run while one warp is waiting on IO, SFU, DPU. Thus it is often best to have more threads than thread processors.
- The best number of threads/block depends on the program, but should be a multiple of 32 such as 64, 128, 192, 256, 768.
- The grid size should be at least the number of multiprocessors, and also works well as a multiple of the number of multiprocessors.
- If __syncthreads() slows the code, use more, smaller blocks.

Low Latency or High Throughput?



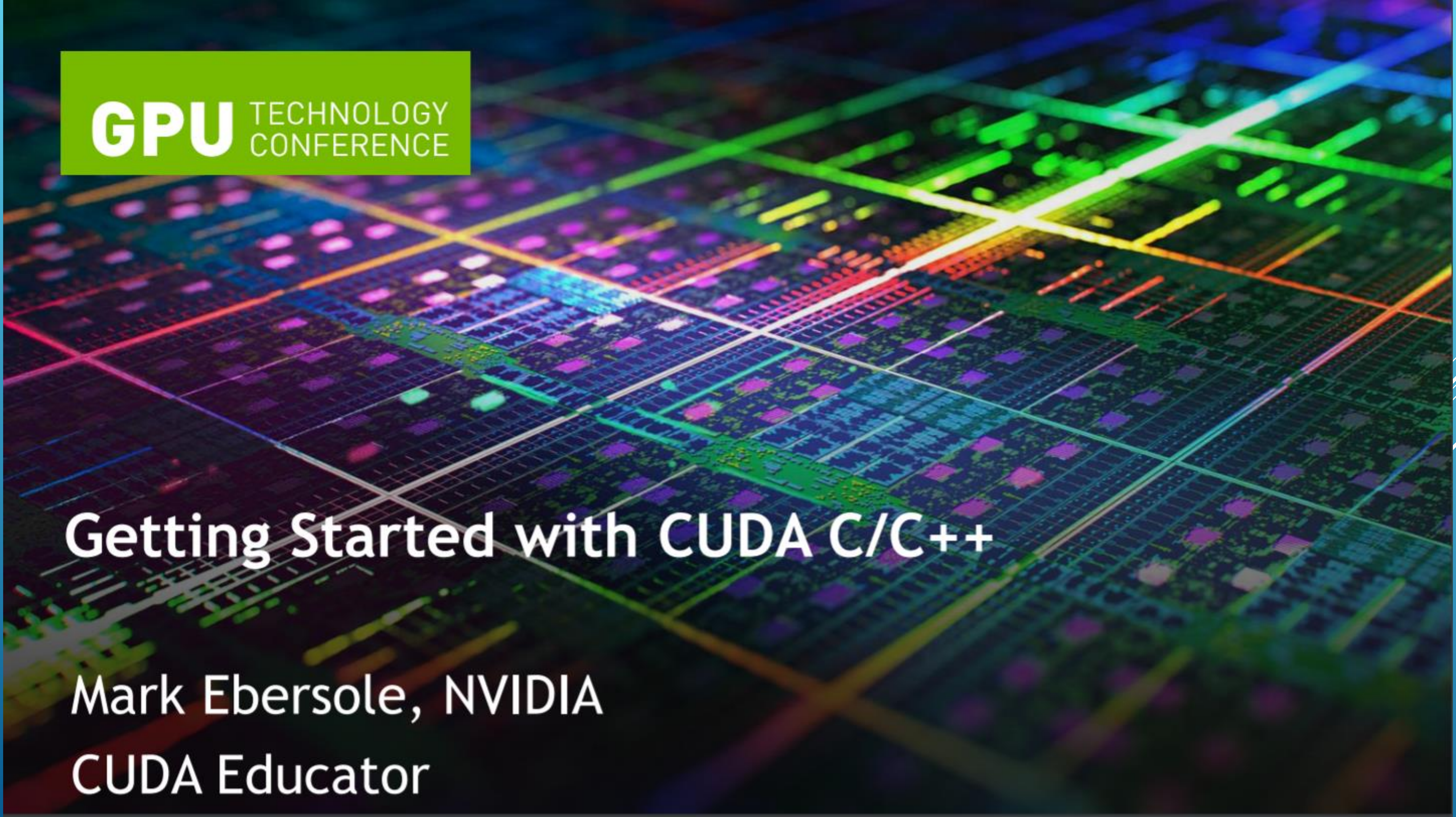
CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation



GPU TECHNOLOGY CONFERENCE

Getting Started with CUDA C/C++

Mark Ebersole, NVIDIA

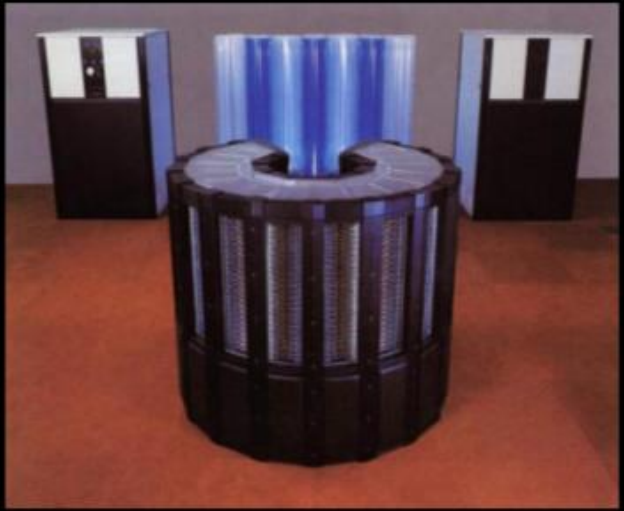
CUDA Educator

Past Massively Parallel Supercomputers

Thinking Machine

Goodyear MPP

MasPar


Cray 2

1.31 TFLOPS on

GeForce 8: First Fully Programmable GPU

Nvidia – Geforce - GeForce é um modelo de aceleradores gráficos 3D para PCs desenvolvido pela NVIDIA.

https://www.nvidia.com/en-us/geforce/

As placas gráficas avançadas, com soluções e tecnologias de games - da NVIDIA.

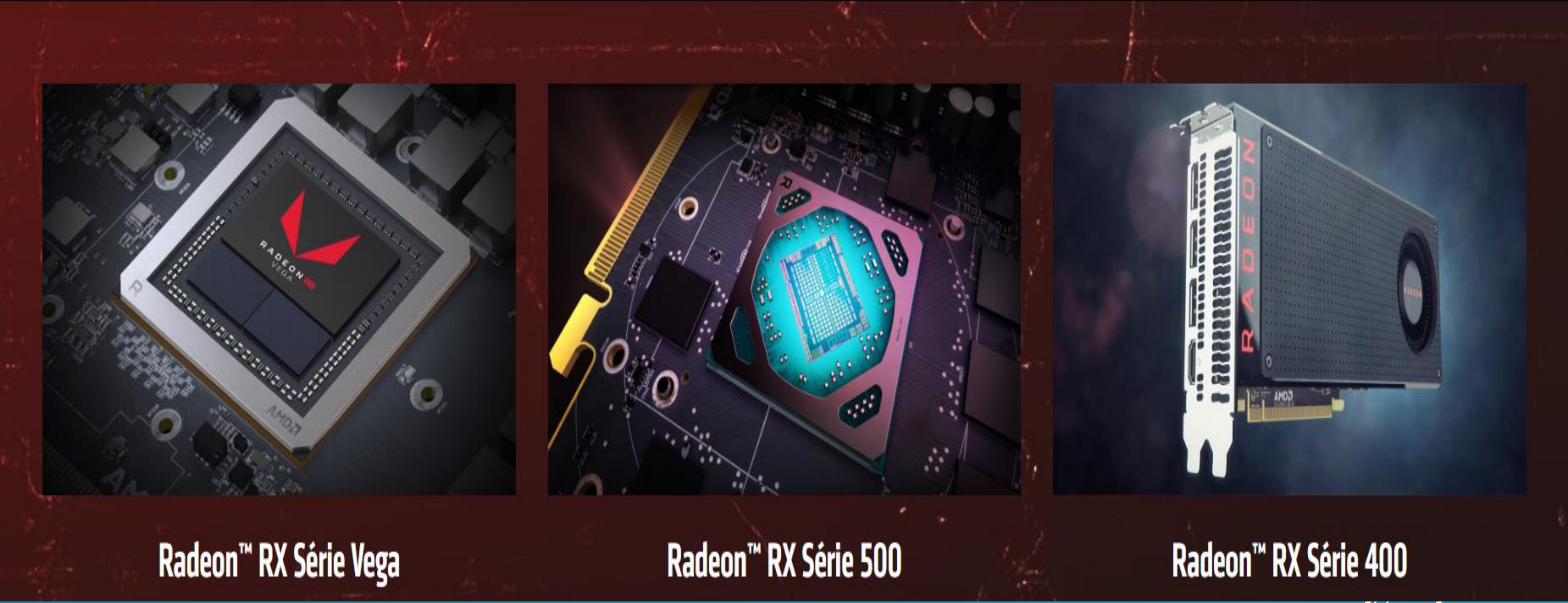
Nvidia – Geforce – Hardware - Notebooks

https://www.geforce.com/hardware/notebook-gpus

Placas de vídeo Radeon™

AMD RADEON VII

A primeira GPU de 7nm para jogos do mundo



https://www.amd.com/pt/graphics/radeon-rx-graphics

INTEL GPUs

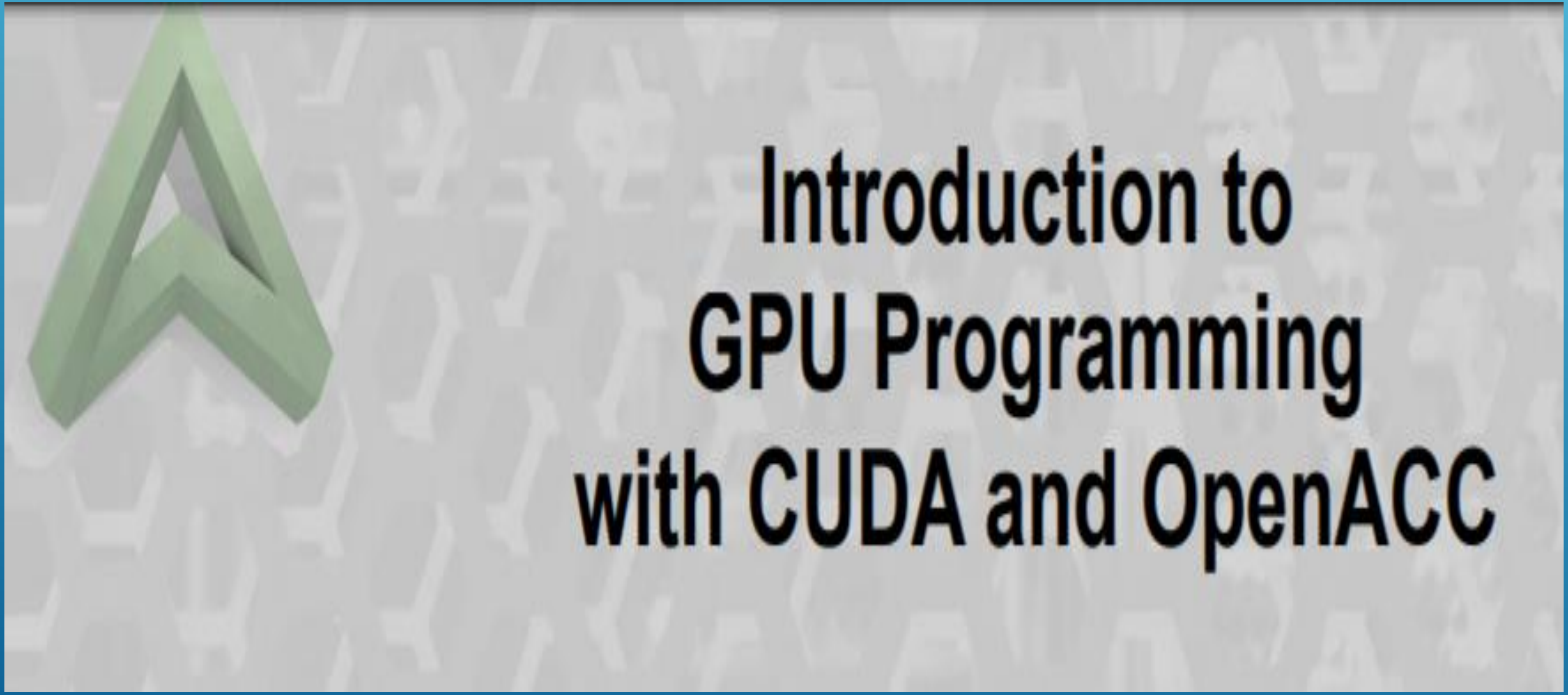
https://laptoping.com/gpus/product/intel-hd-620-review-graphics-of-7th-gen-core-u-series-kaby-lake-cpus/

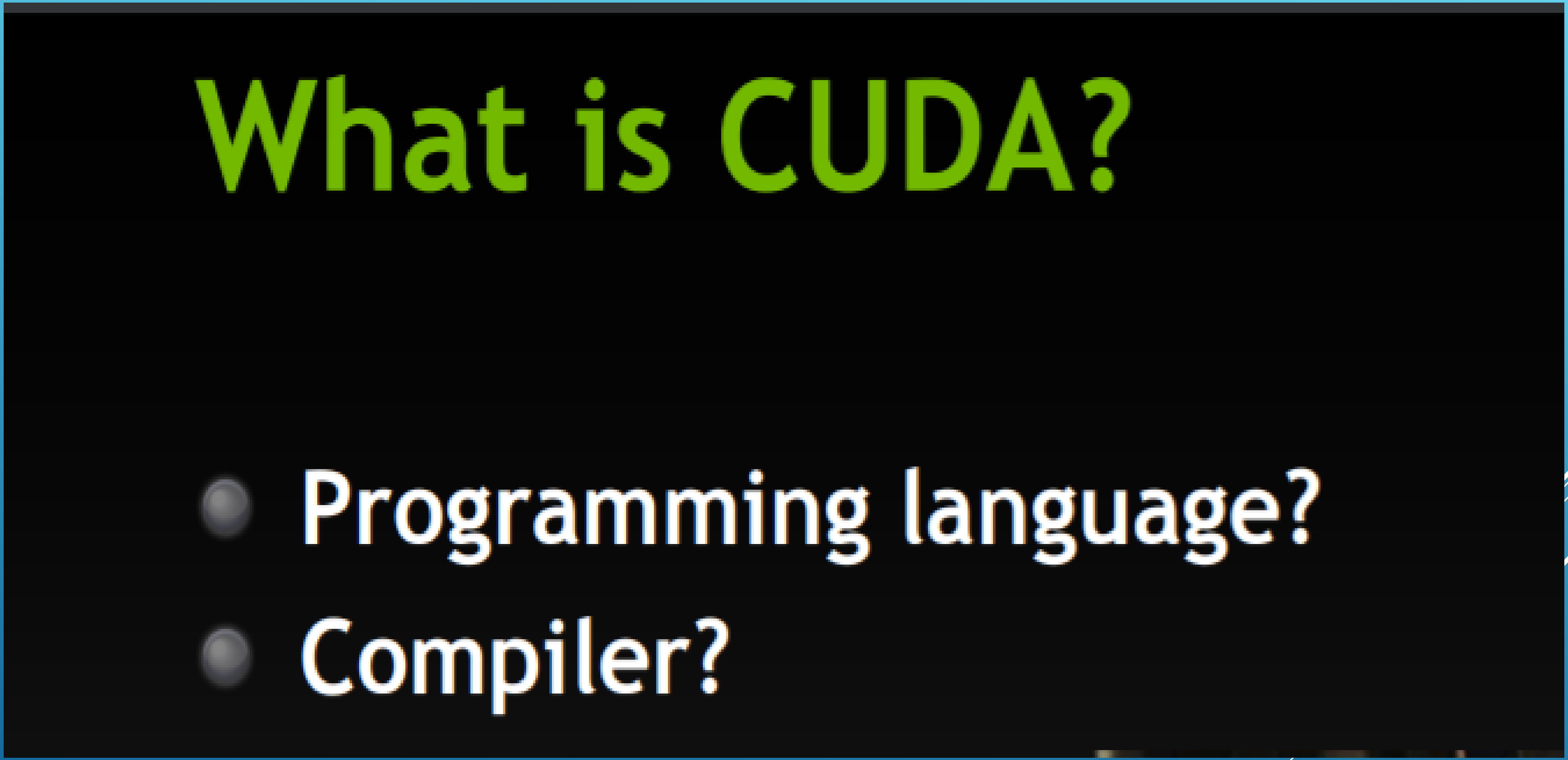
GPU COMPARISON

https://videocardz.com/specials/gpu-comparison

Introduction to CUDA C/C++

Mark Ebersole, NVIDIA
CUDA Educator


Introduction to
GPU Programming
with CUDA and OpenACC



What is CUDA?

- Programming language?
- Compiler?

CUDA Parallel Computing Platform

www.nvidia.com/getcuda

NVIDIA

Programming Approaches

- Libraries: "Drop-in" Acceleration
- OpenACC Directives: Easily Accelerate Apps
- Programming Languages: Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler Tool Chain

LLVM COMPILER INFRASTRUCTURE

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

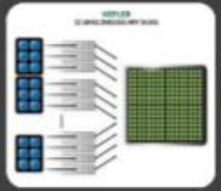
Hardware Capabilities

SMX

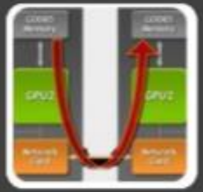
CONTROL LOGIC

Dynamic Parallelism

HyperQ



GPUDirect



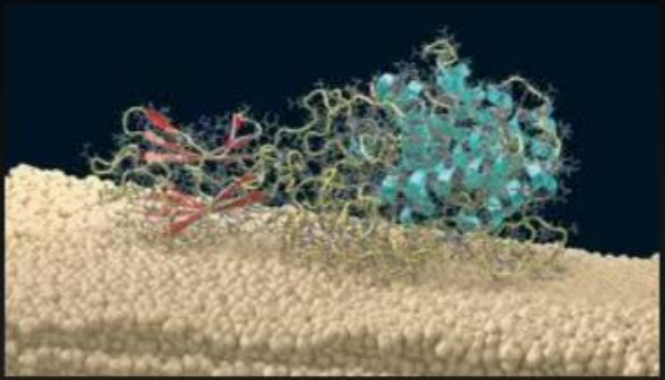
Getting Started with CUDA

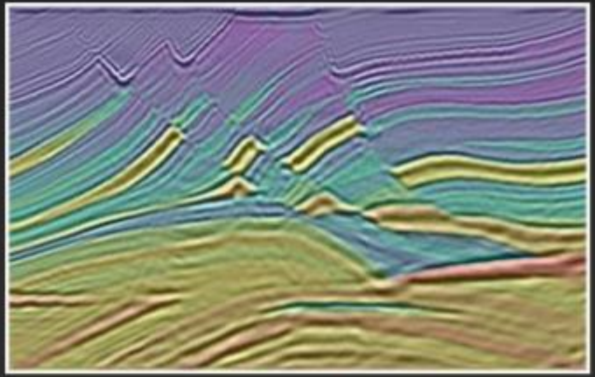
GPU Accelerated Science Applications

Over 145+ Accelerated science apps in our catalog. Just a few:

AMBER

GROMACS


LAMMPS

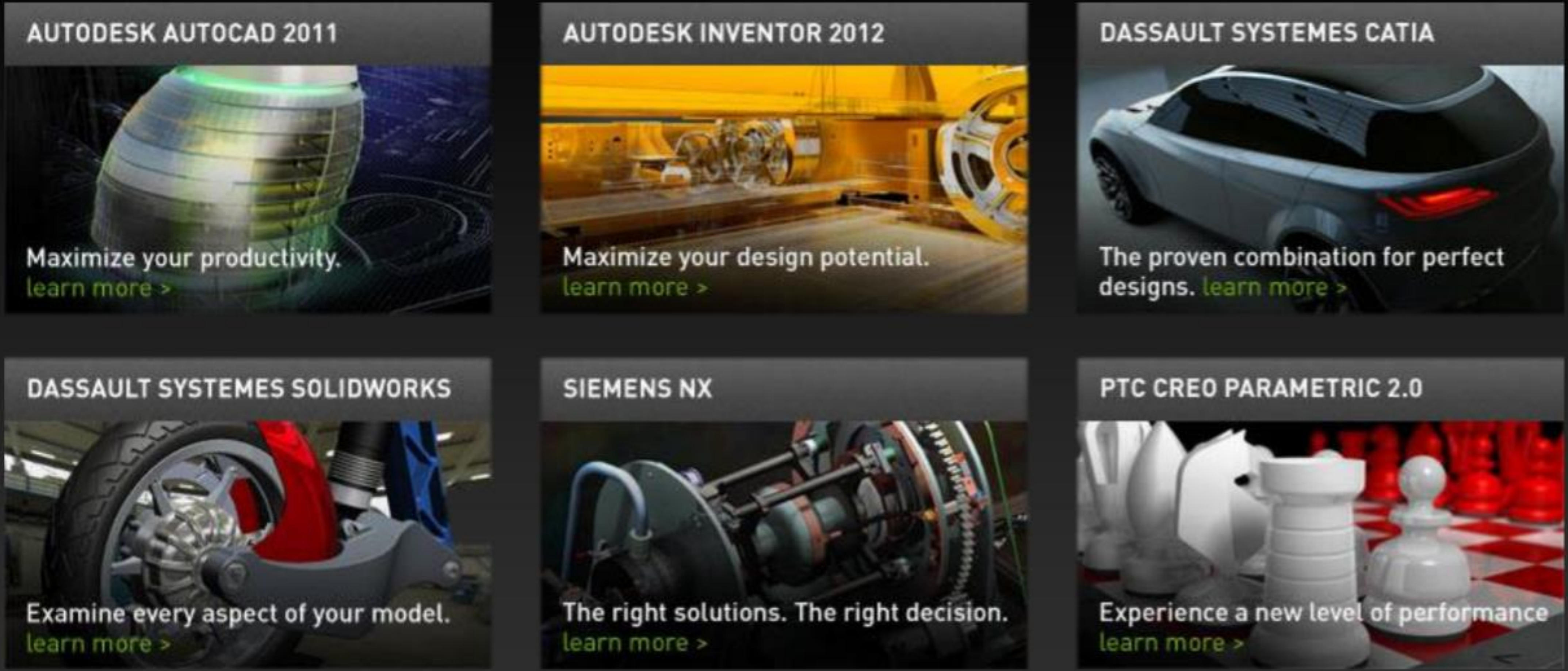

Tsunami RTM

www.nvidia.com/teslaapps

NWChem

GPU Accelerated Workstation Applications

Fifty accelerated workstation apps in our catalog. Just a few:



AUTODESK AUTOCAD 2011

Maximize your productivity.
learn more >

AUTODESK INVENTOR 2012

Maximize your design potential.
learn more >

DASSAULT SYSTEMES CATIA

The proven combination for perfect designs. learn more >

DASSAULT SYSTEMES SOLIDWORKS

Examine every aspect of your model.
learn more >

SIEMENS NX

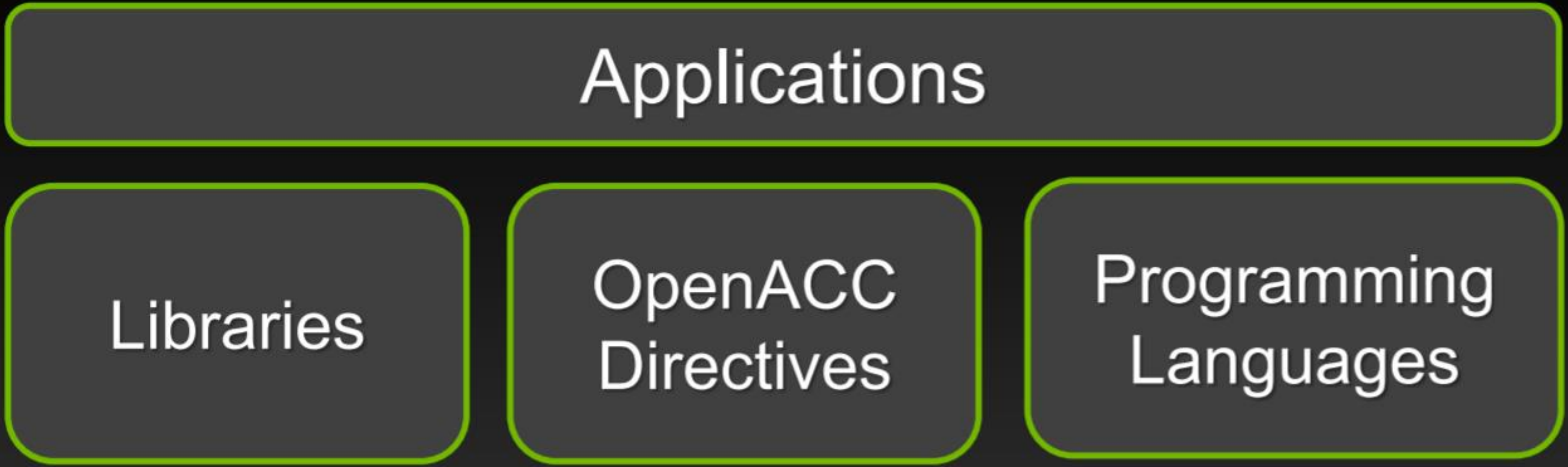
The right solutions. The right decision.
learn more >

PTC CREO PARAMETRIC 2.0

Experience a new level of performance
learn more >

www.nvidia.com/object/gpu-accelerated-applications.html

3 Ways to Accelerate Applications



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

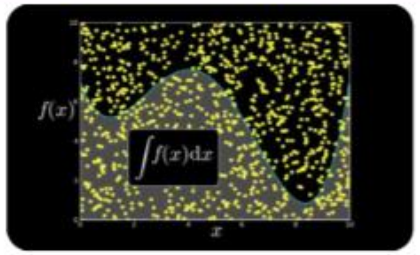
3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL
Vector Signal Image Processing

CULA tools
GPU Accelerated Linear Algebra

MAGMA
Matrix Algebra on GPU and Multicore
open source initiative

NVIDIA cuFFT

ROGUE WAVE SOFTWARE
IMSL Library

ArrayFire Matrix Computations

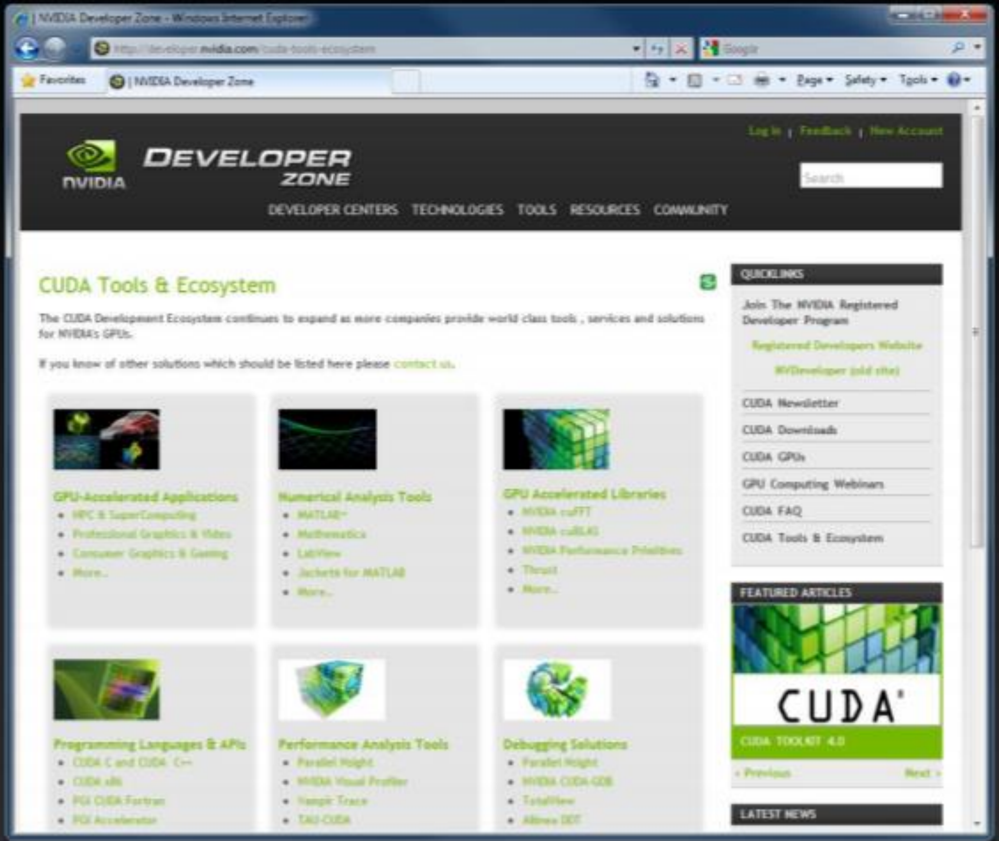
CUSP
Sparse Linear Algebra
open source initiative

Thrust
C++ STL Features for CUDA
open source initiative

Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:
  developer.nvidia.com/cuda-tools-ecosystem

- Watch past GTC library talks



| NVIDIA Developer Zone - Windows Internet Explorer

http://developer.nvidia.com/cuda-tools-ecosystem

CUDA Tools & Ecosystem

The CUDA Development Ecosystem continues to expand as more companies provide world class tools , services and solutions for NVIDIA's GPUs.

If you know of other solutions which should be listed here please contact us.

GPU-Accelerated Applications
- HPC & SuperComputing
- Professional Graphics & Video
- Consumer Graphics & Gaming
- More...

Numerical Analysis Tools
- MATLAB™
- Mathematica
- LabView
- Jackets for MATLAB
- More...

GPU Accelerated Libraries
- NVIDIA cuFFT
- NVIDIA cuBLAS
- NVIDIA Performance Primitives
- Thrust
- More...

Programming Languages & APIs
- CUDA C and CUDA C++
- CUDA x86
- PGI CUDA Fortran
- PGI Accelerator

Performance Analysis Tools
- Parallel Nsight
- NVIDIA Visual Profiler
- Vampir Trace
- TAU CUDA

Debugging Solutions
- Parallel Nsight
- NVIDIA CUDA-GDB
- TotalView
- Allinea DDT

QUICKLINKS

Join The NVIDIA Registered Developer Program

Registered Developers Website

NVDeveloper (old site)

CUDA Newsletter

CUDA Downloads

CUDA GPUs

GPU Computing Webinars

CUDA FAQ

CUDA Tools & Ecosystem

FEATURED ARTICLES

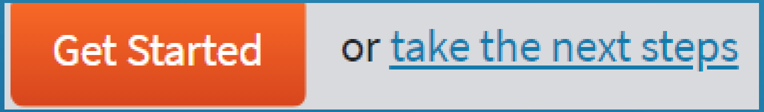
3 Ways to Accelerate Applications

Applications

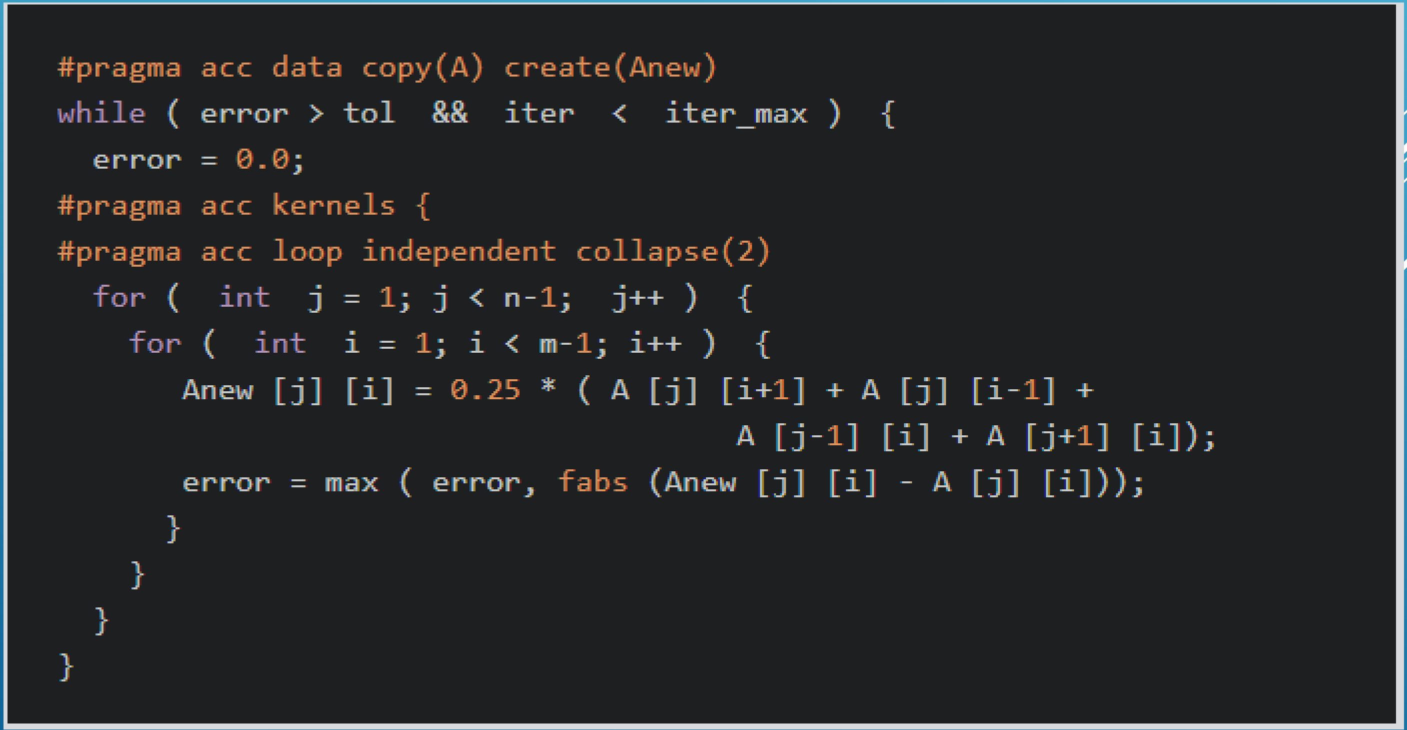
Libraries	OpenACC Directives	Programming Languages
“Drop-in” Acceleration	Easily Accelerate Applications	Maximum Flexibility

What is OpenACC?

OpenACC is a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.

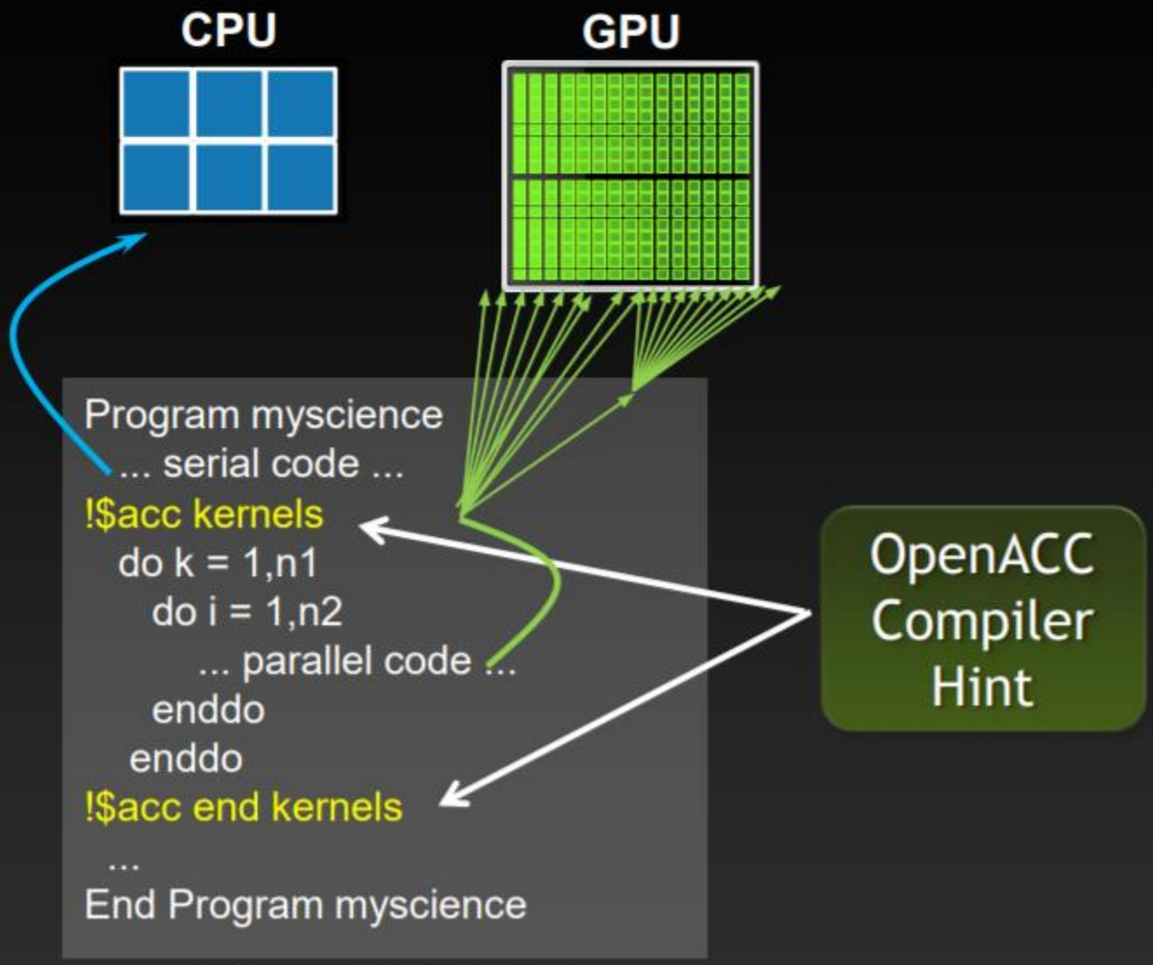


https://www.openacc.org/get-started



```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                          A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
  }
}
```

OpenACC Directives



```
Program myscience
 ... serial code ...
!$acc kernels
 do k = 1,n1
   do i = 1,n2
      ... parallel code ...
   enddo
  enddo
!$acc end kernels
 ...
End Program myscience
```

Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

  www.openacc.org

- OpenACC 2.0 Specification just announced
- Implementations available now from PGI, Cray (beta), and CAPS

The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

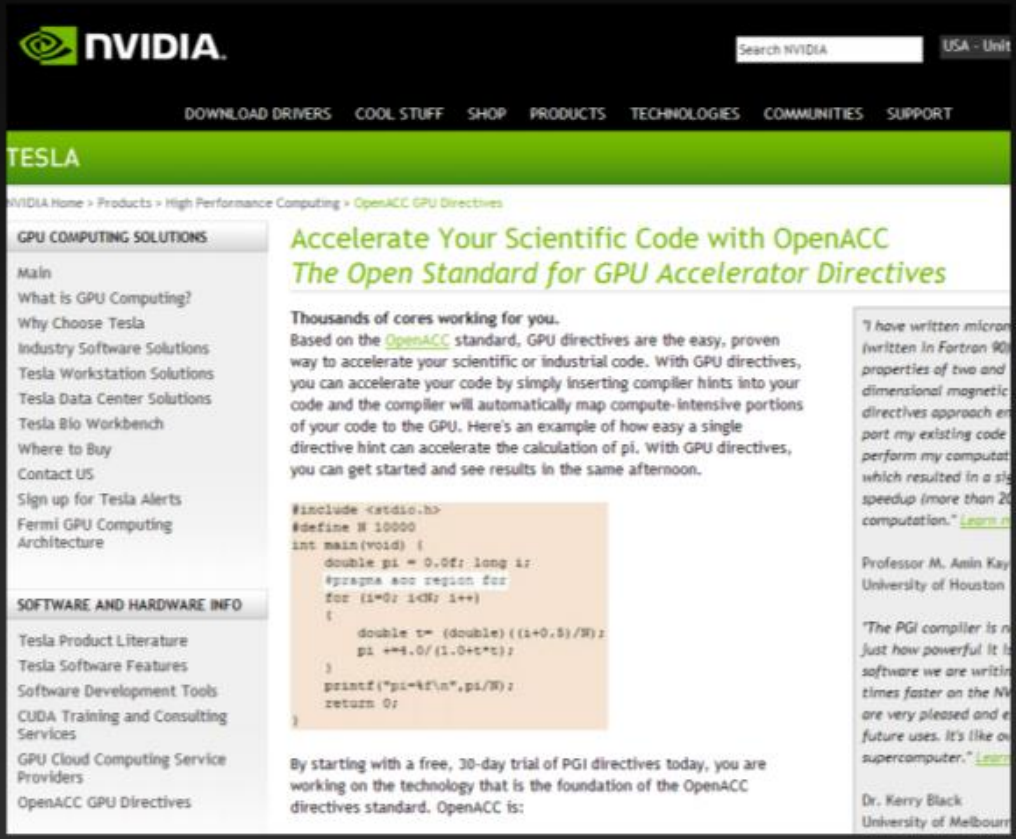
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator

Tools for quick ramp

www.nvidia.com/gpudirectives


NVIDIA
Search NVIDIA
USA - Unit
DOWNLOAD DRIVERS
COOL STUFF
SHOP
PRODUCTS
TECHNOLOGIES
COMMUNITIES
SUPPORT
TESLA
VIDIA Home > Products > High Performance Computing > OpenACC GPU Directives
GPU COMPUTING SOLUTIONS
Main
What is GPU Computing?
Why Choose Tesla
Industry Software Solutions
Tesla Workstation Solutions
Tesla Data Center Solutions
Tesla Bio Workbench
Where to Buy
Contact US
Sign up for Tesla Alerts
Fermi GPU Computing Architecture
SOFTWARE AND HARDWARE INFO
Tesla Product Literature
Tesla Software Features
Software Development Tools
CUDA Training and Consulting Services
GPU Cloud Computing Service Providers
OpenACC GPU Directives

Accelerate Your Scientific Code with OpenACC

The Open Standard for GPU Accelerator Directives

Thousands of cores working for you.
Based on the OpenACC standard, GPU directives are the easy, proven way to accelerate your scientific or industrial code. With GPU directives, you can accelerate your code by simply inserting compiler hints into your code and the compiler will automatically map compute-intensive portions of your code to the GPU. Here's an example of how easy a single directive hint can accelerate the calculation of pi. With GPU directives, you can get started and see results in the same afternoon.

```
#include <stdio.h>
#define N 10000
int main(void) {
    double pi = 0.0f; long i;
    #pragma acc region for
    for (i=0; i<N; i++)
    {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

By starting with a free, 30-day trial of PGI directives today, you are working on the technology that is the foundation of the OpenACC directives standard. OpenACC is:

"I have written micron (written in Fortran 90) properties of two and dimensional magnetic directives approach en port my existing code perform my computat which resulted in a sig speedup (more than 20 computation." Learn m

Professor M. Amin Kay
University of Houston

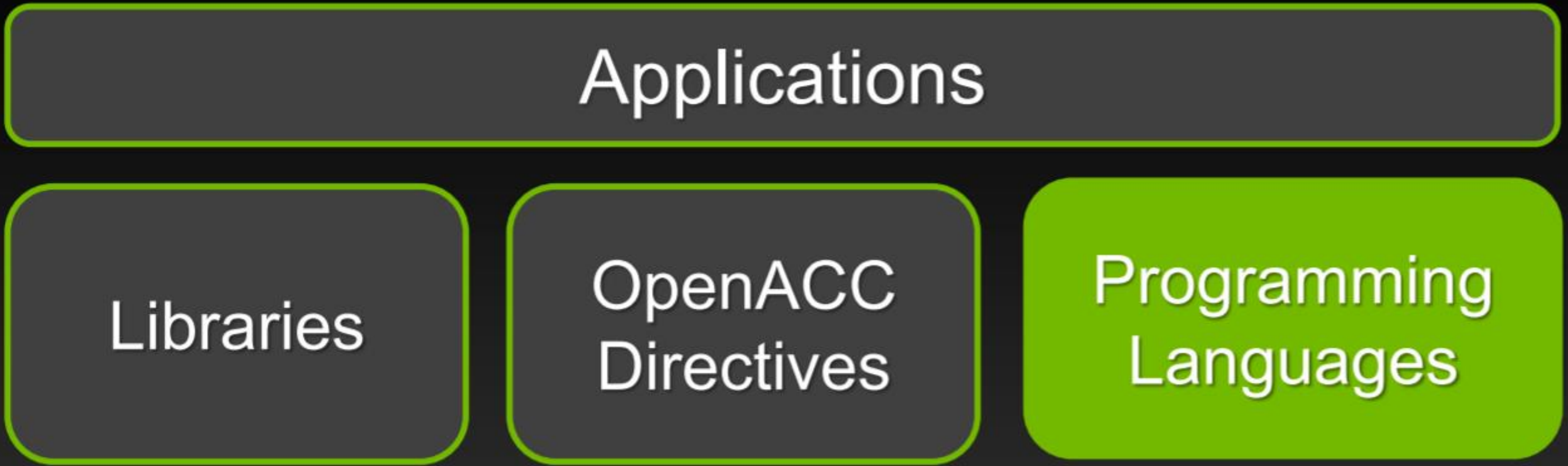
"The PGI compiler is n just how powerful it is software we are writin times faster on the NV are very pleased and e future uses. It's like ou supercomputer." Learn

Dr. Kerry Black
University of Melbourn

NVIDIA

PGI®

3 Ways to Accelerate Applications



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead
C#	GPU.NET

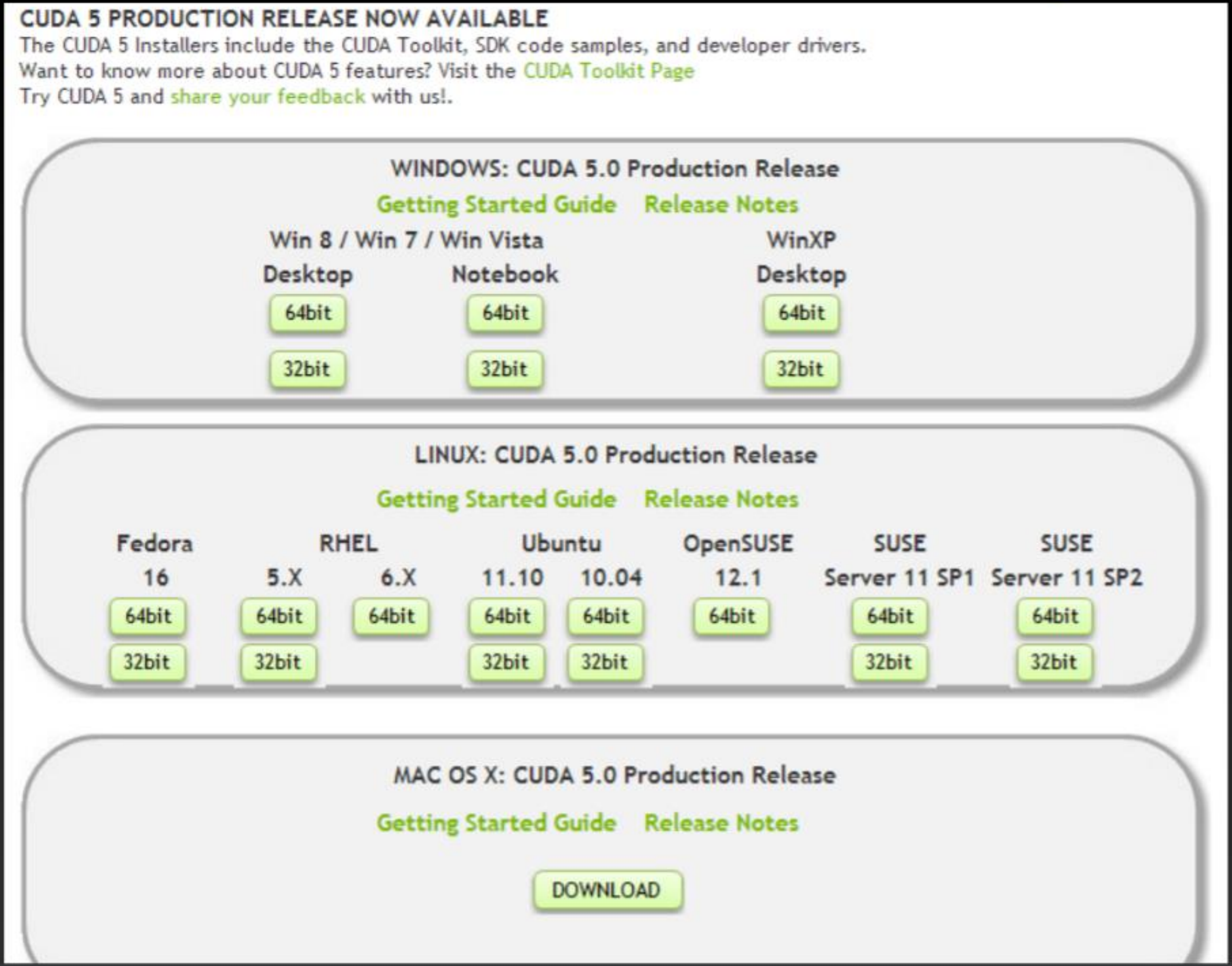
Programming a CUDA Language

- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Prerequisites

- You (probably) need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

CUDA 5 Toolkit and SDK - www.nvidia.com/getcuda



CUDA 5 PRODUCTION RELEASE NOW AVAILABLE
The CUDA 5 Installers include the CUDA Toolkit, SDK code samples, and developer drivers.
Want to know more about CUDA 5 features? Visit the CUDA Toolkit Page
Try CUDA 5 and share your feedback with us!.

WINDOWS: CUDA 5.0 Production Release

Getting Started Guide   Release Notes

Win 8 / Win 7 / Win Vista Desktop	Win 8 / Win 7 / Win Vista Notebook	WinXP Desktop
64bit	64bit	64bit
32bit	32bit	32bit

LINUX: CUDA 5.0 Production Release

Getting Started Guide   Release Notes

Fedora 16	RHEL 5.X	RHEL 6.X	Ubuntu 11.10	Ubuntu 10.04	OpenSUSE 12.1	SUSE Server 11 SP1	SUSE Server 11 SP2
64bit	64bit	64bit	64bit	64bit	64bit	64bit	64bit
32bit	32bit		32bit	32bit		32bit	32bit

MAC OS X: CUDA 5.0 Production Release

Getting Started Guide   Release Notes

DOWNLOAD

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing

SAXPY

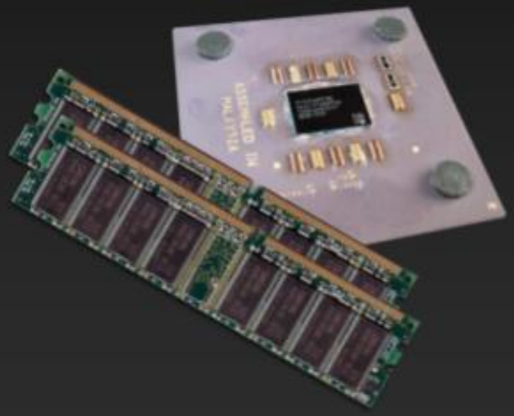
Standard C Code

```
void saxpy(int n, float a, float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0f, x, y);
```

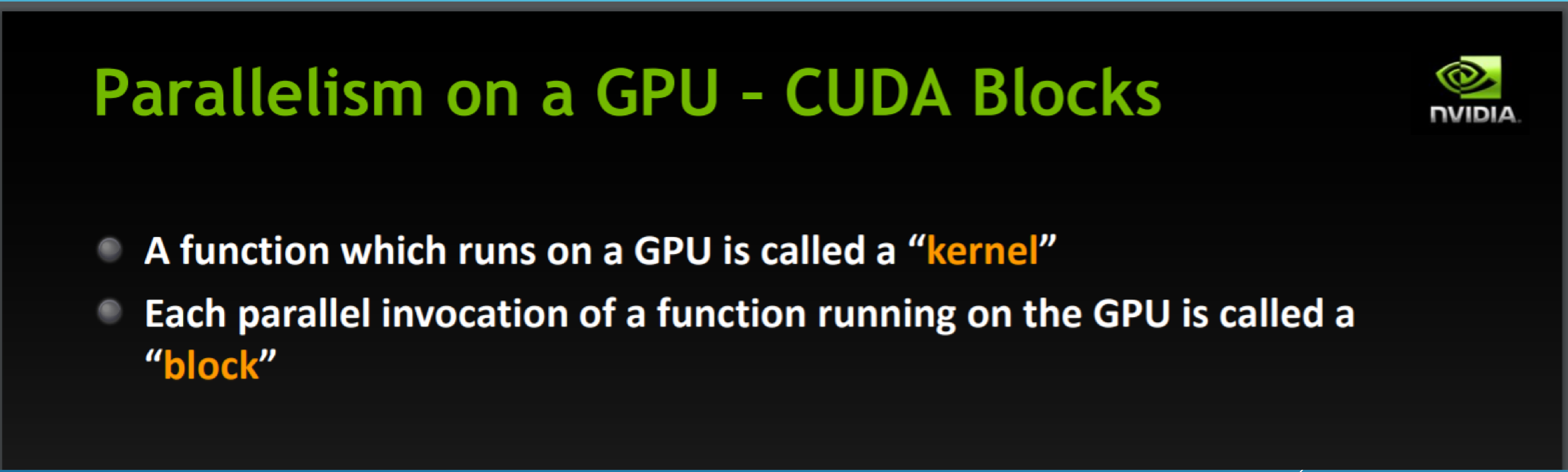
http://developer.nvidia.com/cuda-toolkit

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)


Host

Device



Parallelism on a GPU – CUDA Blocks

- A function which runs on a GPU is called a “kernel”
- Each parallel invocation of a function running on the GPU is called a “block”

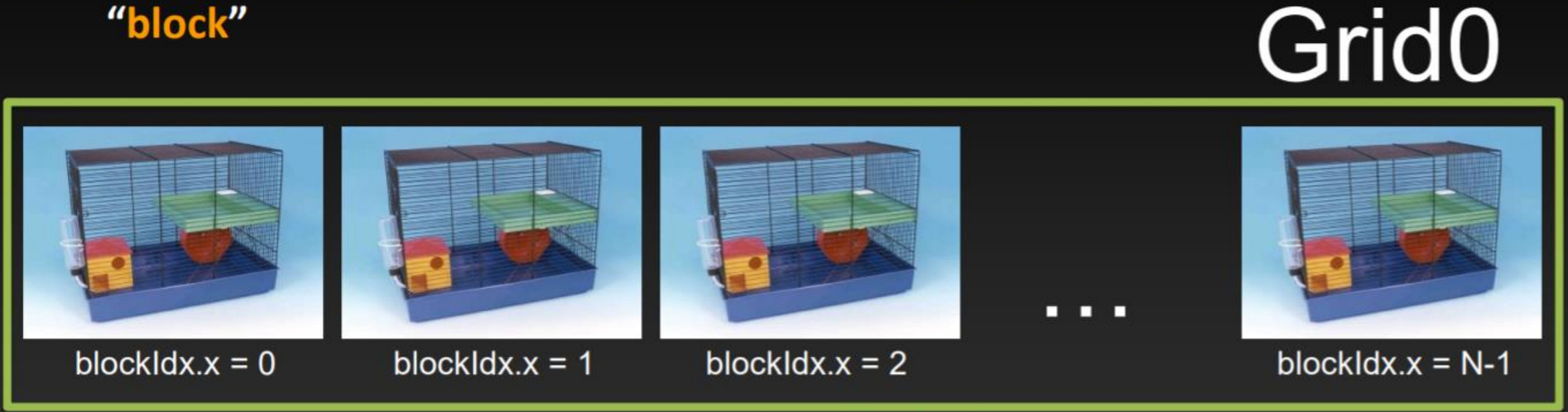
Parallelism on a GPU – CUDA Blocks

- el”
- the GPU is called a

= BLOCK

Parallelism on a GPU – CUDA Blocks

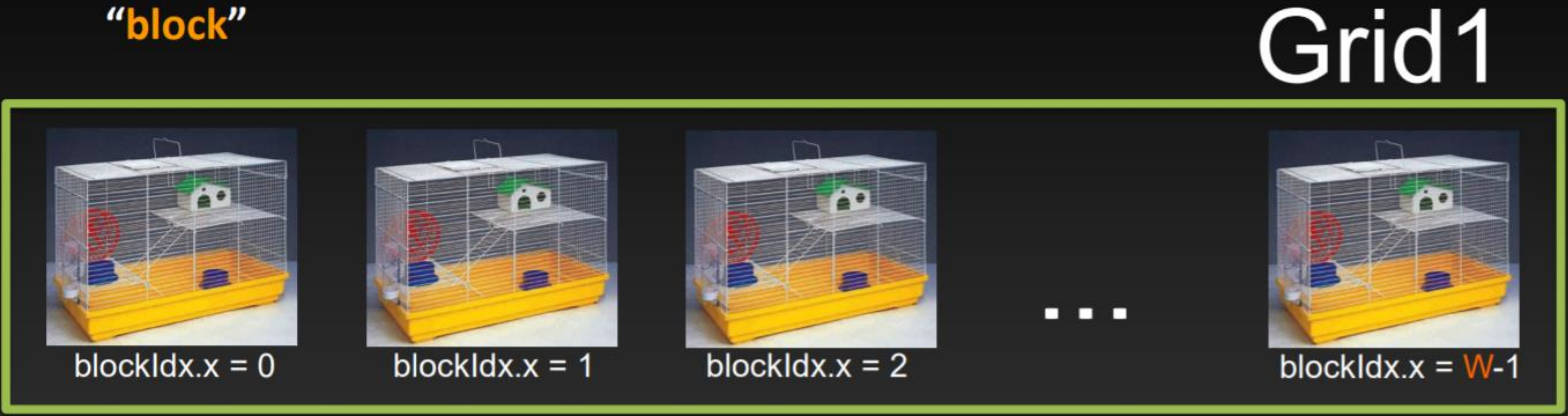
- A function which runs on a GPU is called a "**kernel**"
- Each parallel invocation of a function running on the GPU is called a "**block**"



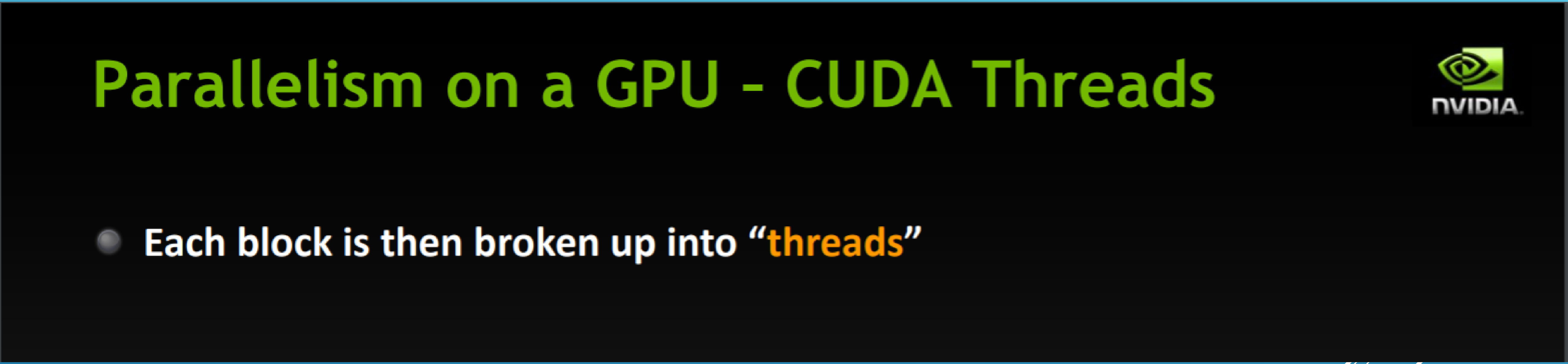
- A block can identify itself by reading **blockIdx.x**

Parallelism on a GPU – CUDA Blocks

- A function which runs on a GPU is called a “**kernel**”
- Each parallel invocation of a function running on the GPU is called a “**block**”



- A block can identify itself by reading **blockIdx.x**



Parallelism on a GPU – CUDA Threads

- Each block is then broken up into “threads”

Pa CUDA Threads

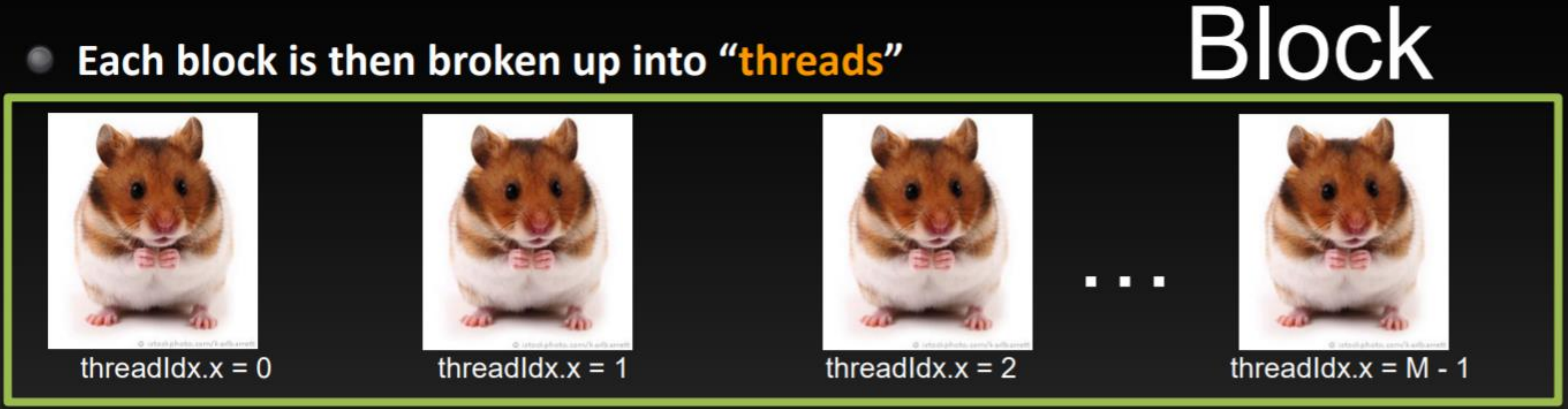
- E reads"

= THREAD

- A threadIdx.x
- T can be read with blockDim.x

Parallelism on a GPU – CUDA Threads

- Each block is then broken up into "threads"



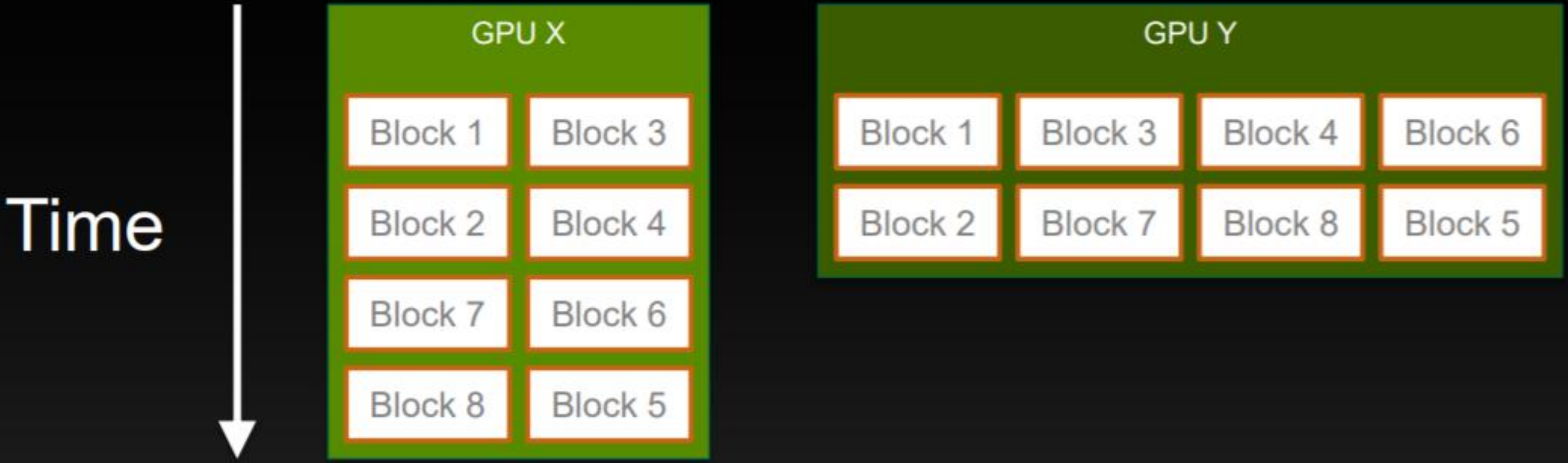
- A thread can identify itself by reading threadIdx.x
- The total number of threads per block can be read with blockDim.x
  - In the above example blockDim.x = M

Why threads and blocks?

- Threads within a block can
  - Communicate very quickly (share memory)
  - Synchronize (wait for all threads to catch up)

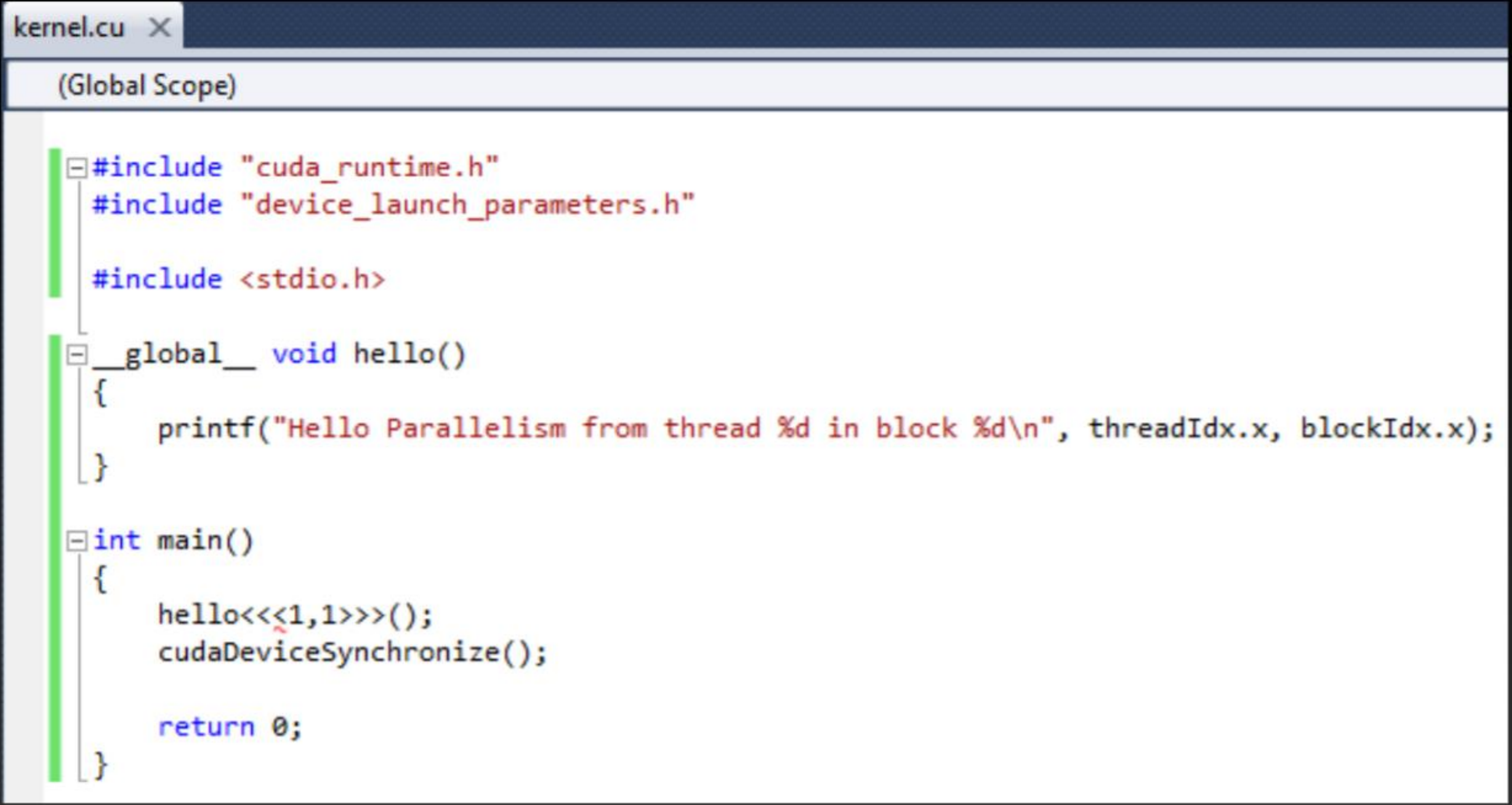
- Why break up into blocks?
  - A block cannot be broken up among multiple SMs (streaming multiprocessors), and you want to keep all SMs busy.
  - Allows the HW to scale the number of blocks running in parallel based on GPU capability

Why threads and blocks?

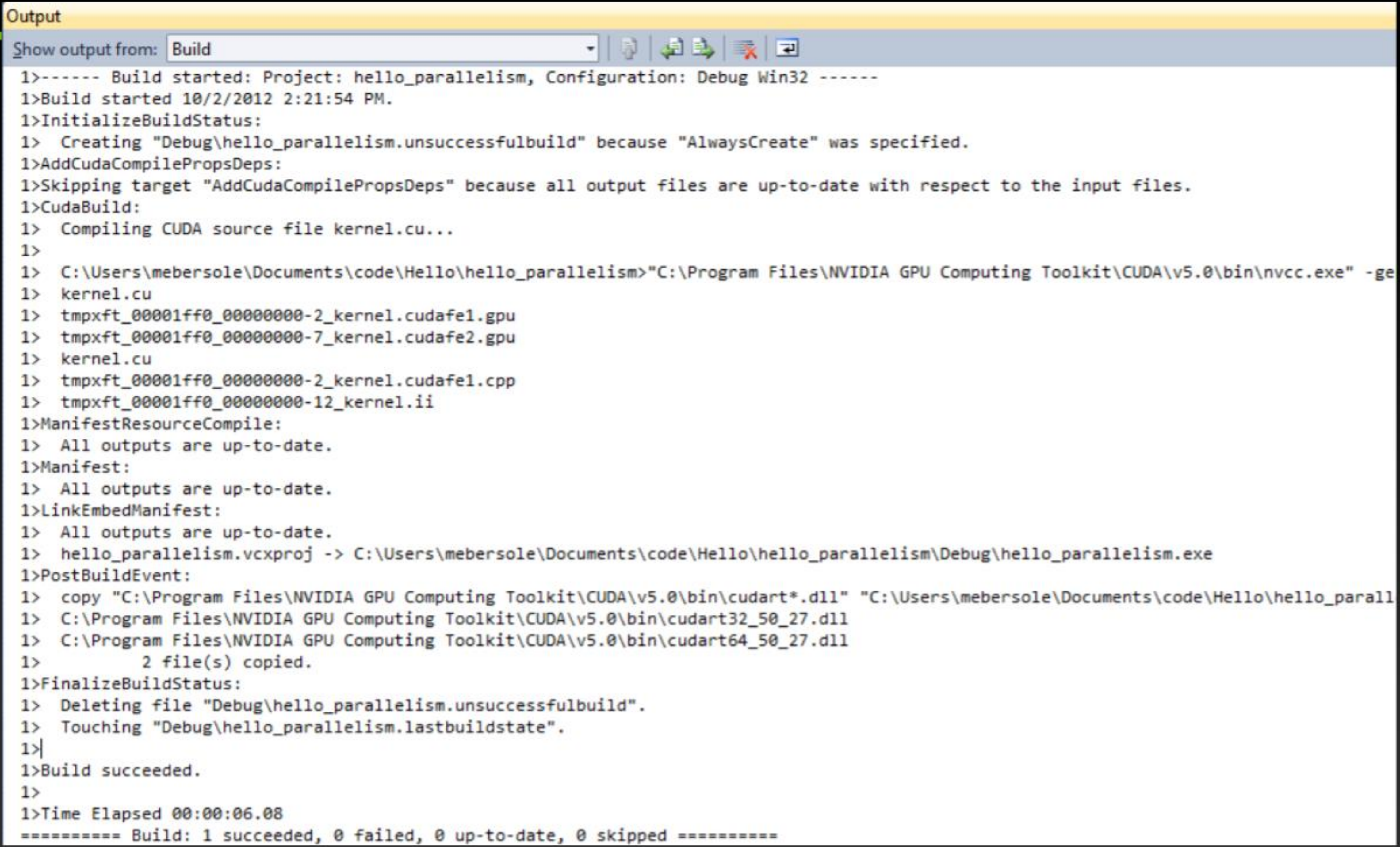


- **Why break up into blocks?**
  - **A block cannot be broken up among multiple SMs (streaming multiprocessors), and you want to keep all SMs busy.**
  - **Allows the HW to scale the number of blocks running in parallel based on GPU capability**

Hello Parallelism!



```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

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

int main()
{
    hello<<<1,1>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

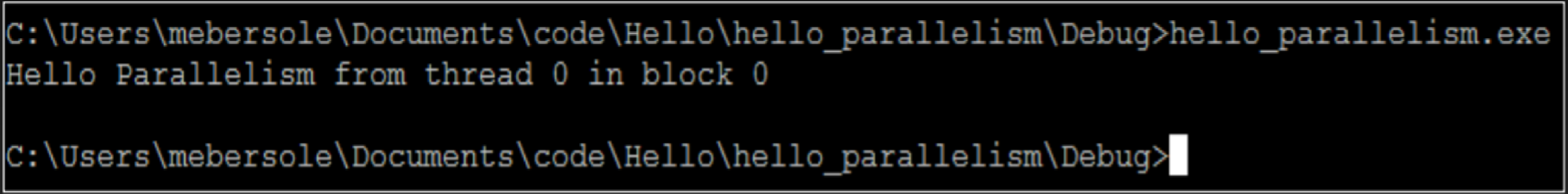


Output

Show output from: Build

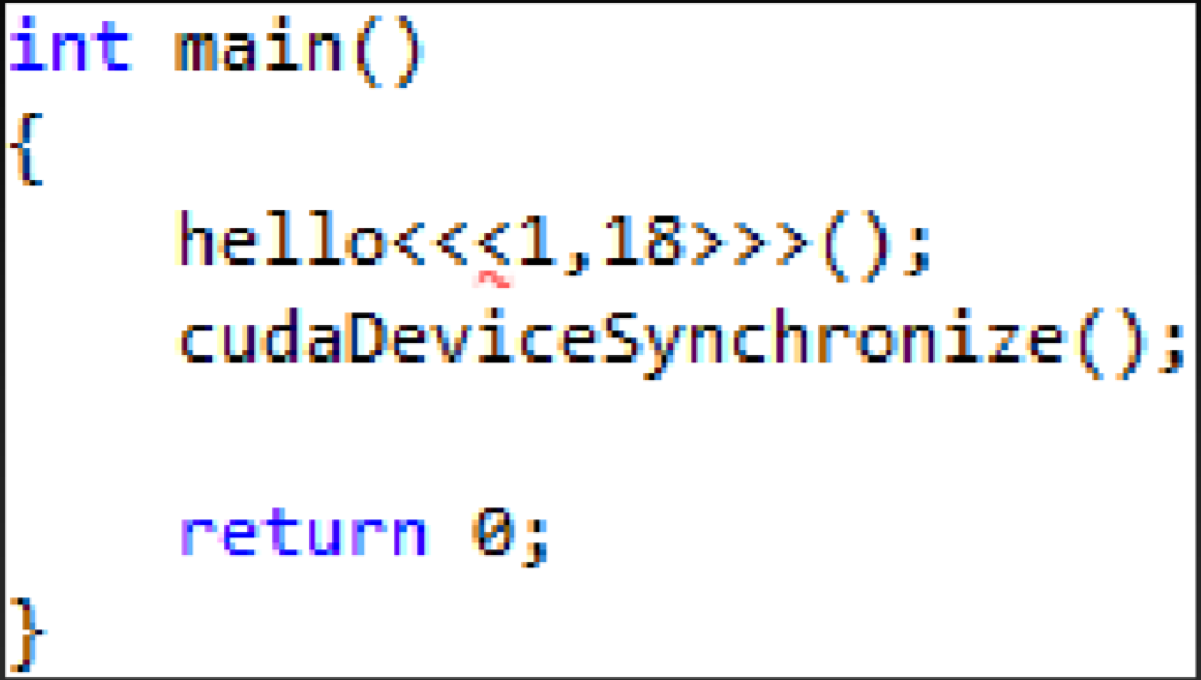
```
1>------ Build started: Project: hello_parallelism, Configuration: Debug Win32 ------
1>Build started 10/2/2012 2:21:54 PM.
1>InitializeBuildStatus:
1>  Creating "Debug\hello_parallelism.unsuccessfulbuild" because "AlwaysCreate" was specified.
1>AddCudaCompilePropsDeps:
1>Skipping target "AddCudaCompilePropsDeps" because all output files are up-to-date with respect to the input files.
1>CudaBuild:
1>  Compiling CUDA source file kernel.cu...
1>
1>  C:\Users\mebersole\Documents\code\Hello\hello_parallelism>"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin\nvcc.exe" -ge
1>  kernel.cu
1>  tmpxft_00001ff0_00000000-2_kernel.cudafe1.gpu
1>  tmpxft_00001ff0_00000000-7_kernel.cudafe2.gpu
1>  kernel.cu
1>  tmpxft_00001ff0_00000000-2_kernel.cudafe1.cpp
1>  tmpxft_00001ff0_00000000-12_kernel.ii
1>ManifestResourceCompile:
1>  All outputs are up-to-date.
1>Manifest:
1>  All outputs are up-to-date.
1>LinkEmbedManifest:
1>  All outputs are up-to-date.
1>  hello_parallelism.vcxproj -> C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug\hello_parallelism.exe
1>PostBuildEvent:
1>  copy "C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin\cudart*.dll" "C:\Users\mebersole\Documents\code\Hello\hello_parall
1>  C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin\cudart32_50_27.dll
1>  C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin\cudart64_50_27.dll
1>          2 file(s) copied.
1>FinalizeBuildStatus:
1>  Deleting file "Debug\hello_parallelism.unsuccessfulbuild".
1>  Touching "Debug\hello_parallelism.lastbuildstate".
1>
1>Build succeeded.
1>
1>Time Elapsed 00:00:06.08
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Hello Parallelism!



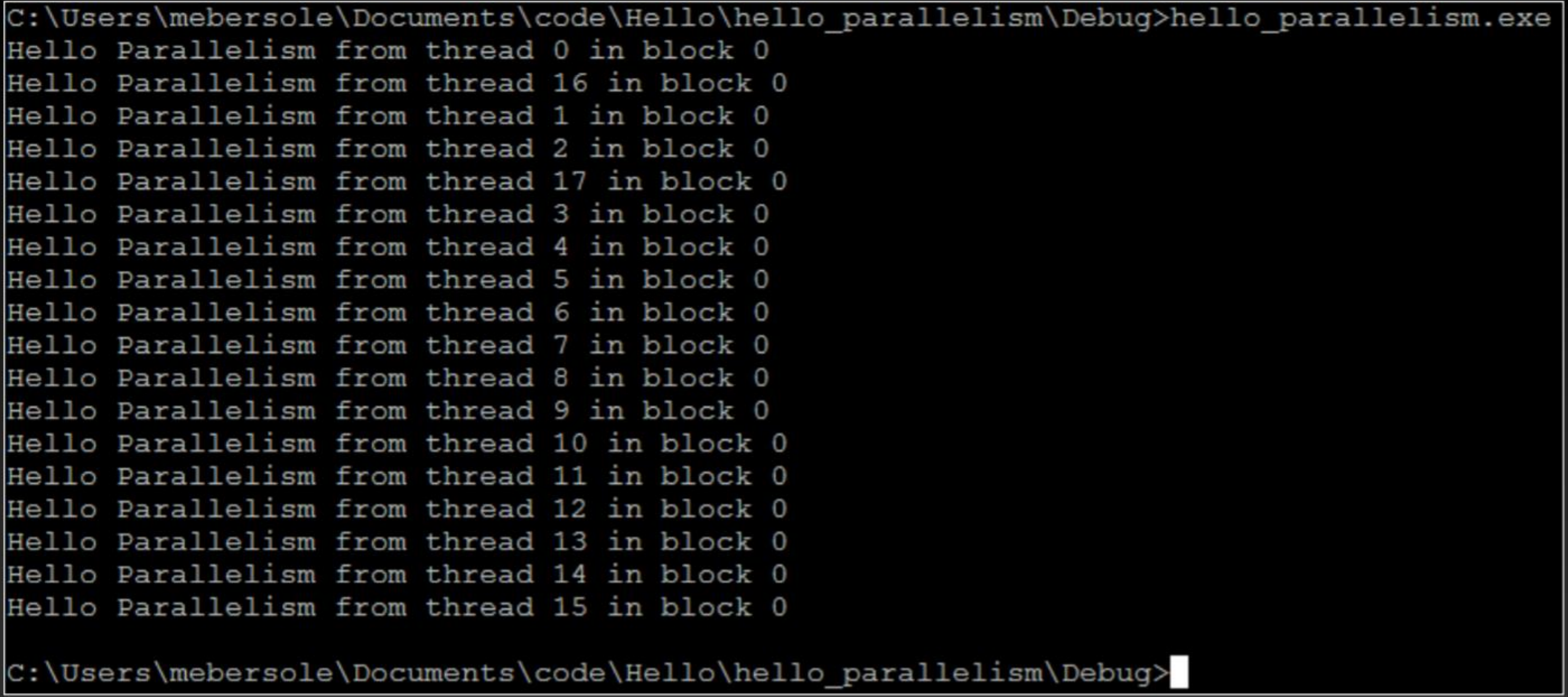
```
C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug>hello_parallelism.exe
Hello Parallelism from thread 0 in block 0

C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug>
```

Hello Parallelism!



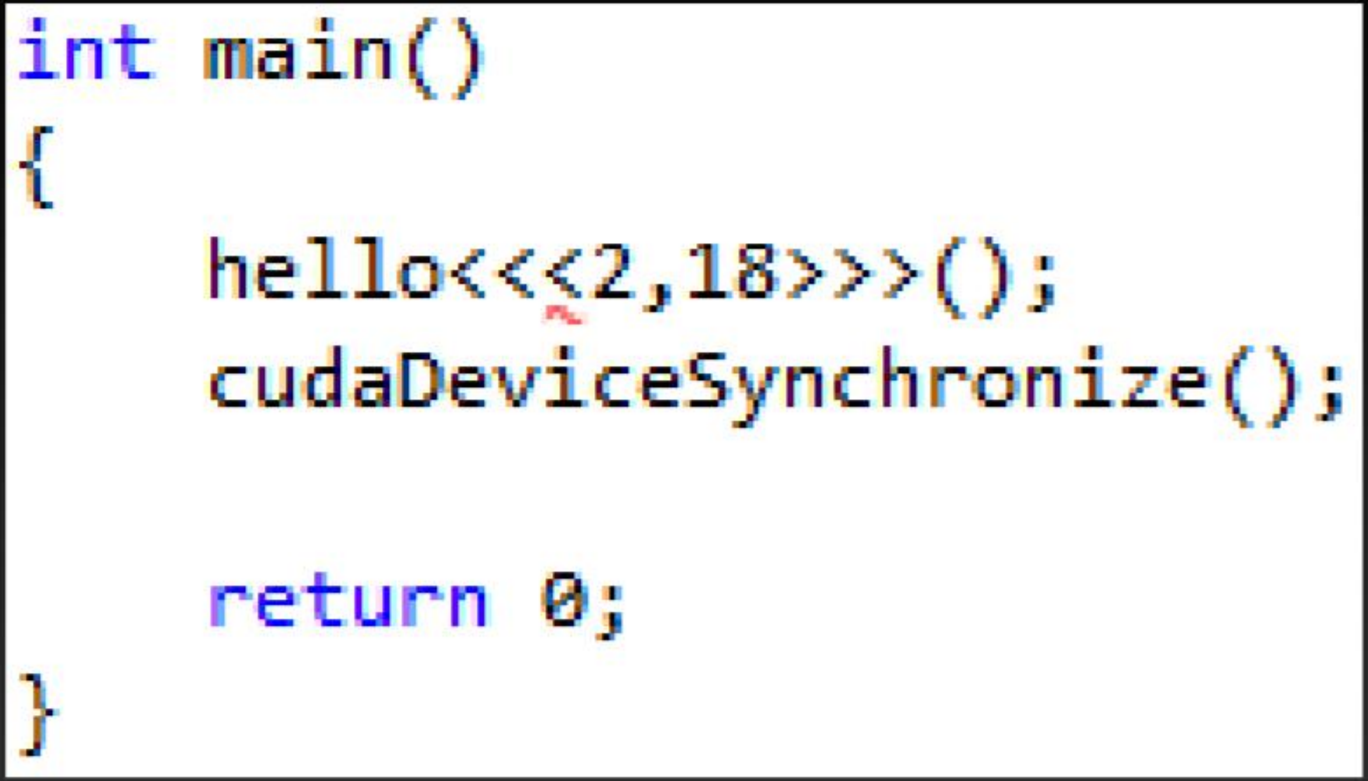
```
int main()
{
    hello<<<1,18>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

Hello Parallelism



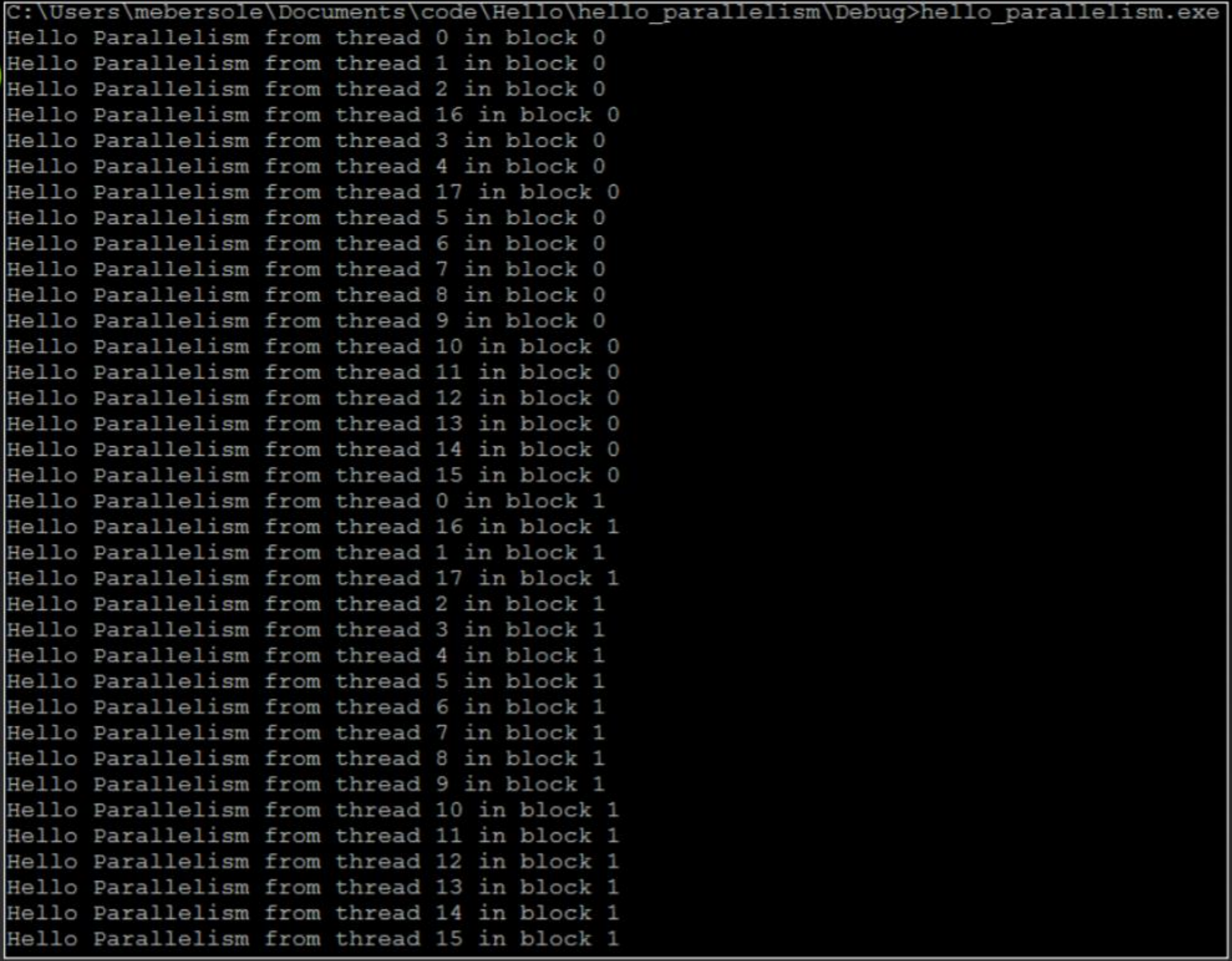
```
C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug>hello_parallelism.exe
Hello Parallelism from thread 0 in block 0
Hello Parallelism from thread 16 in block 0
Hello Parallelism from thread 1 in block 0
Hello Parallelism from thread 2 in block 0
Hello Parallelism from thread 17 in block 0
Hello Parallelism from thread 3 in block 0
Hello Parallelism from thread 4 in block 0
Hello Parallelism from thread 5 in block 0
Hello Parallelism from thread 6 in block 0
Hello Parallelism from thread 7 in block 0
Hello Parallelism from thread 8 in block 0
Hello Parallelism from thread 9 in block 0
Hello Parallelism from thread 10 in block 0
Hello Parallelism from thread 11 in block 0
Hello Parallelism from thread 12 in block 0
Hello Parallelism from thread 13 in block 0
Hello Parallelism from thread 14 in block 0
Hello Parallelism from thread 15 in block 0

C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug>
```

Hello Parallelism!

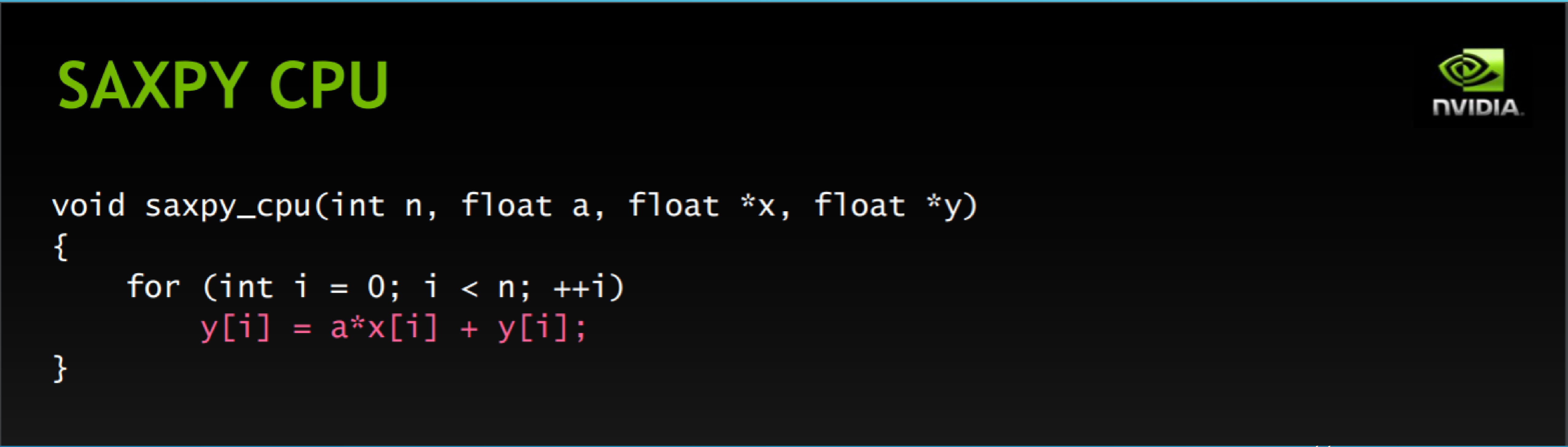


```
int main()
{
    hello<<<2,18>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

Hello

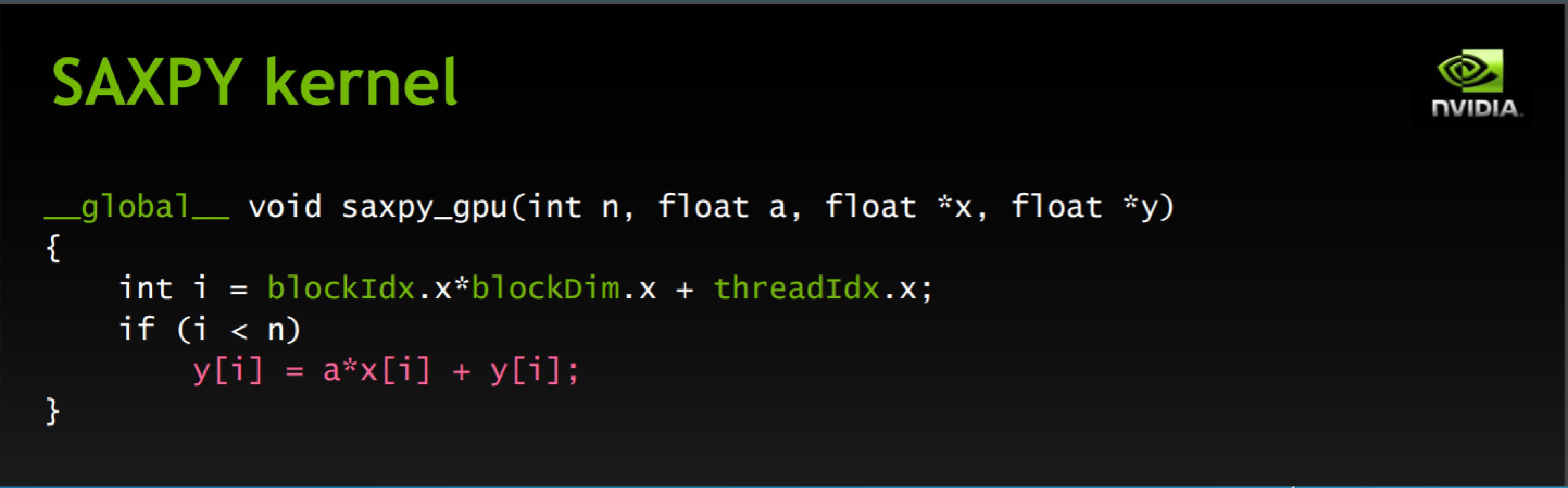


```
C:\Users\mebersole\Documents\code\Hello\hello_parallelism\Debug>hello_parallelism.exe
Hello Parallelism from thread 0 in block 0
Hello Parallelism from thread 1 in block 0
Hello Parallelism from thread 2 in block 0
Hello Parallelism from thread 16 in block 0
Hello Parallelism from thread 3 in block 0
Hello Parallelism from thread 4 in block 0
Hello Parallelism from thread 17 in block 0
Hello Parallelism from thread 5 in block 0
Hello Parallelism from thread 6 in block 0
Hello Parallelism from thread 7 in block 0
Hello Parallelism from thread 8 in block 0
Hello Parallelism from thread 9 in block 0
Hello Parallelism from thread 10 in block 0
Hello Parallelism from thread 11 in block 0
Hello Parallelism from thread 12 in block 0
Hello Parallelism from thread 13 in block 0
Hello Parallelism from thread 14 in block 0
Hello Parallelism from thread 15 in block 0
Hello Parallelism from thread 0 in block 1
Hello Parallelism from thread 16 in block 1
Hello Parallelism from thread 1 in block 1
Hello Parallelism from thread 17 in block 1
Hello Parallelism from thread 2 in block 1
Hello Parallelism from thread 3 in block 1
Hello Parallelism from thread 4 in block 1
Hello Parallelism from thread 5 in block 1
Hello Parallelism from thread 6 in block 1
Hello Parallelism from thread 7 in block 1
Hello Parallelism from thread 8 in block 1
Hello Parallelism from thread 9 in block 1
Hello Parallelism from thread 10 in block 1
Hello Parallelism from thread 11 in block 1
Hello Parallelism from thread 12 in block 1
Hello Parallelism from thread 13 in block 1
Hello Parallelism from thread 14 in block 1
Hello Parallelism from thread 15 in block 1
```



SAXPY CPU

```
void saxpy_cpu(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```



SAXPY kernel

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

SAXPY kernel

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

blockIdx.x:

Our Block ID

blockDim.x:

Number of threads per block

threadIdx.x:

Our thread ID

SAXPY kernel

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

i is now an index into our input and output arrays

SAXPY kernel - with data

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

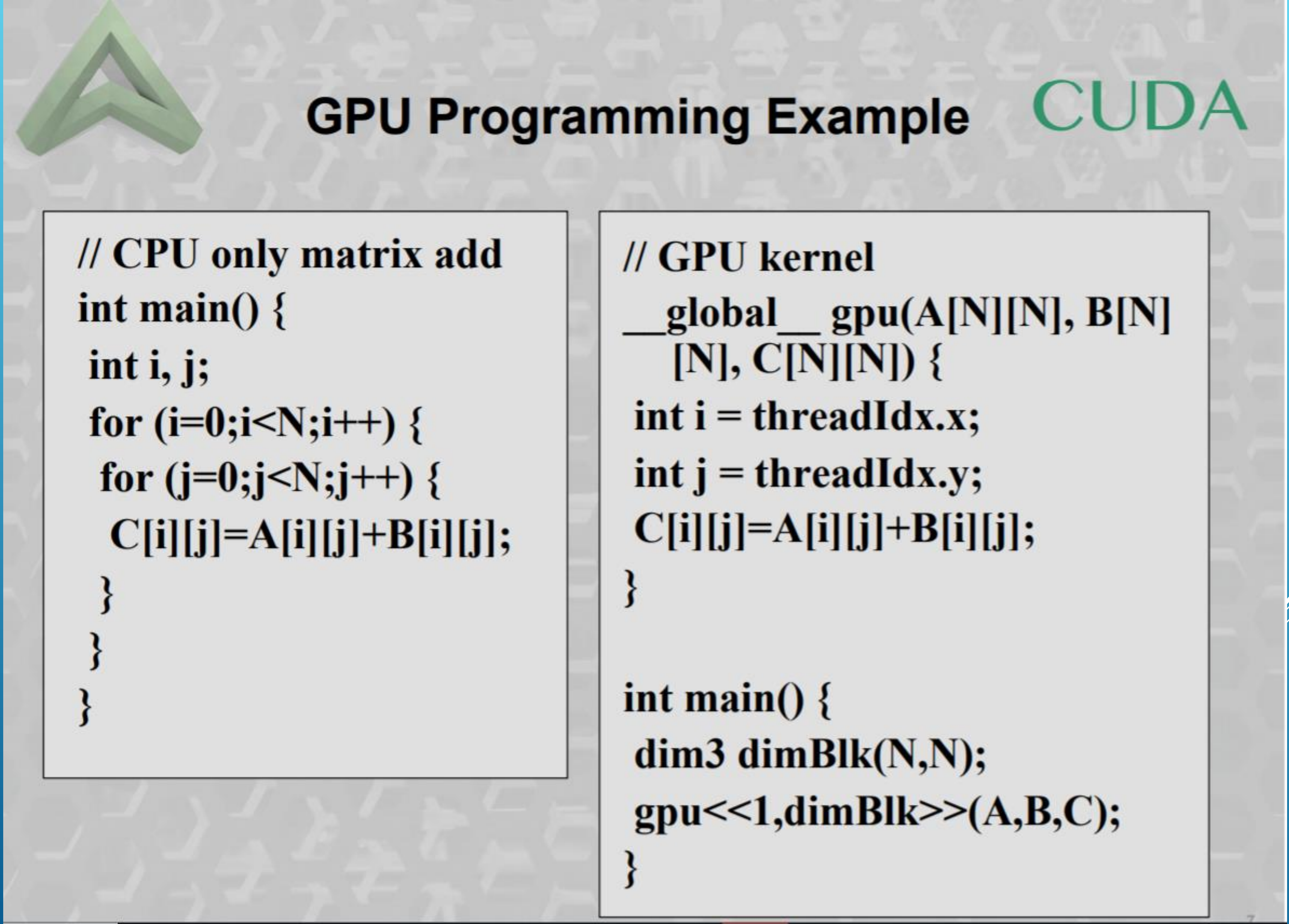
- Let’s work with 30 data elements
    - Broken into 3 blocks, with 10 threads per block
- So, blockDim.x = 10

SAXPY kernel - with data

```
__global__ void saxpy_gpu(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a*x[i] + y[i];
}
```

10 threads (hamsters) each with a different `i`

- For blockIdx.x = 0
  - `i = 0 * 10 + threadIdx.x = {0,1,2,3,4,5,6,7,8,9}`
- For blockIdx.x = 1
  - `i = 1 * 10 + threadIdx.x = {10,11,12,13,14,15,16,17,18,19}`
- For blockIdx.x = 2
  - `i = 2 * 10 + threadIdx.x = {20,21,22,23,24,25,26,27,28,29}`



GPU Programming Example

CUDA

```
// CPU only matrix add
int main() {
 int i, j;
 for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
   C[i][j]=A[i][j]+B[i][j];
  }
 }
}
```

```
// GPU kernel
__global__ gpu(A[N][N], B[N]
    [N], C[N][N]) {
 int i = threadIdx.x;
 int j = threadIdx.y;
 C[i][j]=A[i][j]+B[i][j];
}

int main() {
 dim3 dimBlk(N,N);
 gpu<<1,dimBlk>>(A,B,C);
}
```

Calling saxpy_gpu: `main()`

Standard C Code

```
#define N (2048 * 512)
int main(void) {
    float *x, *y;    // host copies

    int size = N * sizeof(float);




    // Alloc space for host copies of
    // x & y and setup input values
    x = (float *)malloc(size);
    random_floats(x, N);
    y = (float *)malloc(size);
    random_floats(y, N);
```

Parallel C Code

```
#define N (2048 * 512)
int main(void) {
    float *x, *y;    // host copies
    float *d_x, *d_y;// device copies
    int size = N * sizeof(float);

    // Alloc space for device copies
    cudaMalloc((void **)&d_x, size);
    cudaMalloc((void **)&d_y, size);

    // Alloc space for host copies of
    // x & y and setup input values
    x = (float *)malloc(size);
    random_floats(x, N);
    y = (float *)malloc(size);
    random_floats(y, N);
```

Calling saxpy_gpu: `main()`

Standard C Code

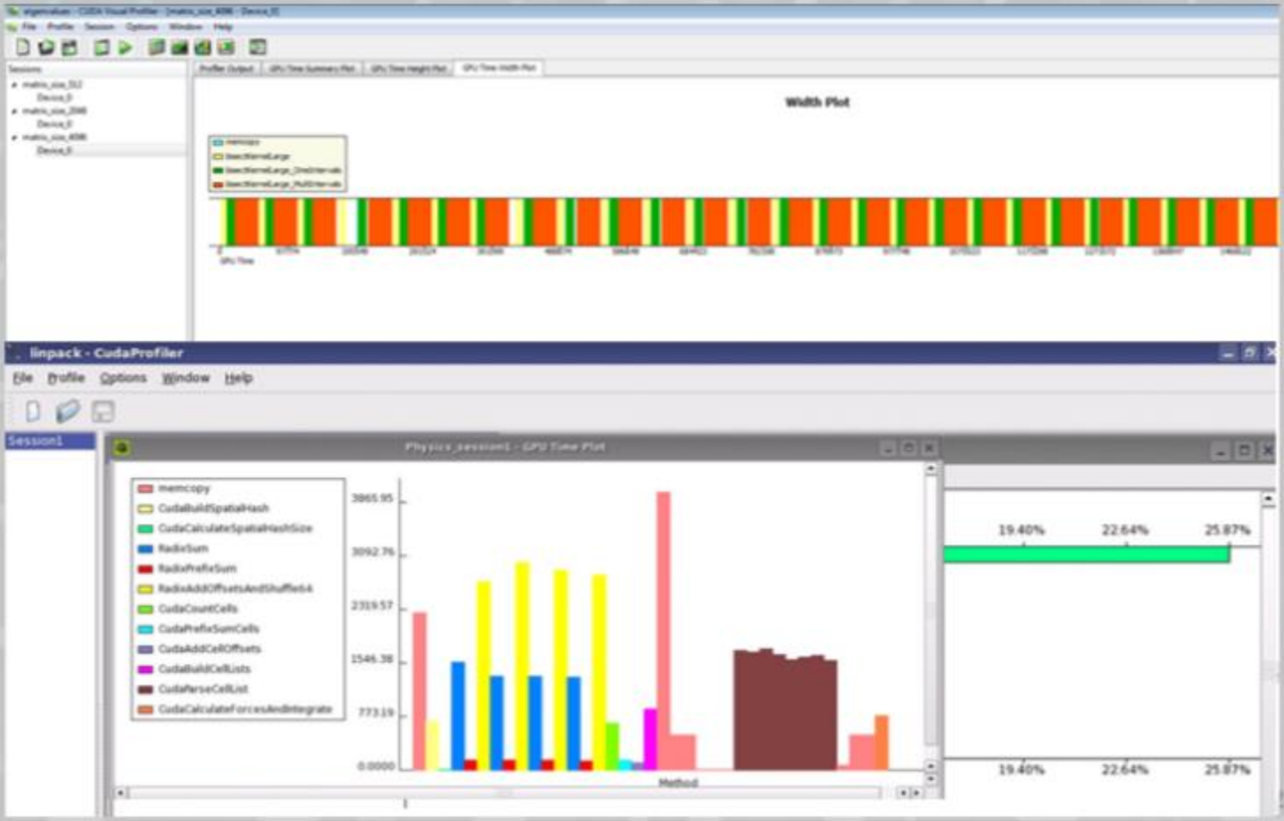
```
    // Launch saxpy on CPU
    saxpy_cpu(N, 2.0f, x, y);


    // Cleanup

    free(x); free(y);
    return 0;
}
```

Parallel C Code

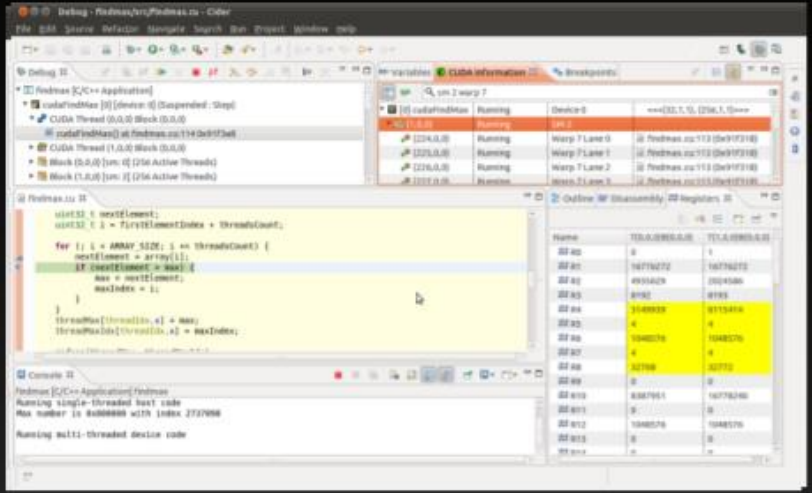
```
    // Copy input to device
    cudaMemcpy(d_x, &x, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, &y, size, cudaMemcpyHostToDevice);

    // Launch saxpy kernel on GPU
    saxpy_gpu<<<4096,256>>>(N, 2.0f, d_x, d_y);

    // Copy result back to host
    cudaMemcpy(&y, d_y, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_x); cudaFree(d_y);
    free(x); free(y);
    return 0;
}
```

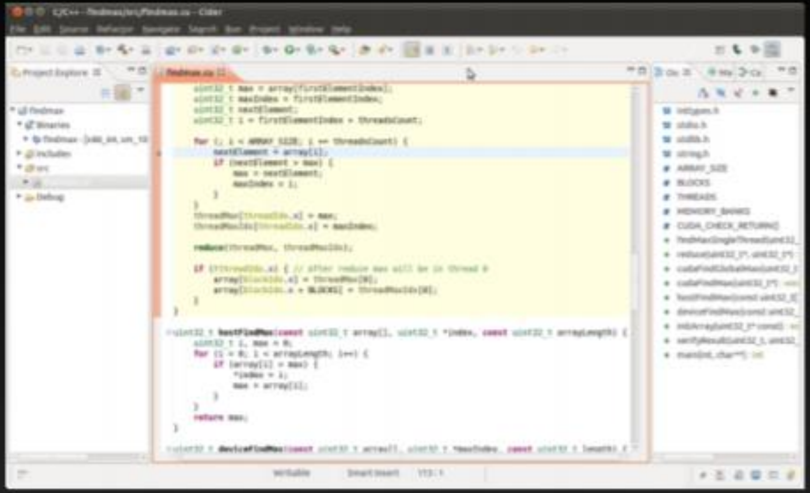
Other CUDA Tools

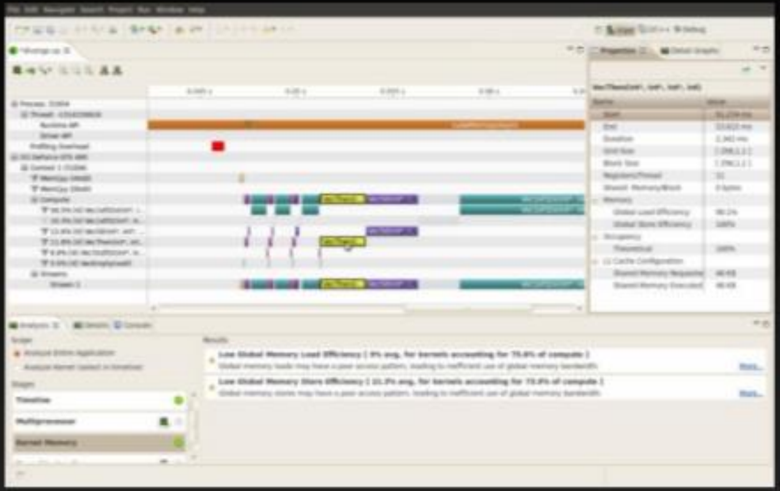
- CUDA Memory Checker (cuda-memcheck) can be used to find memory violations
- CUDA debugger (cuda-gdb) is an extension of the GNU debugger for Linux
- NVIDIA Parallel Nsight is a debugger for Microsoft Visual Studio
- CUDA Visual Profiler



NVIDIA® Nsight™ Eclipse Edition for Linux and MacOS







CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs

Nsight Debugger

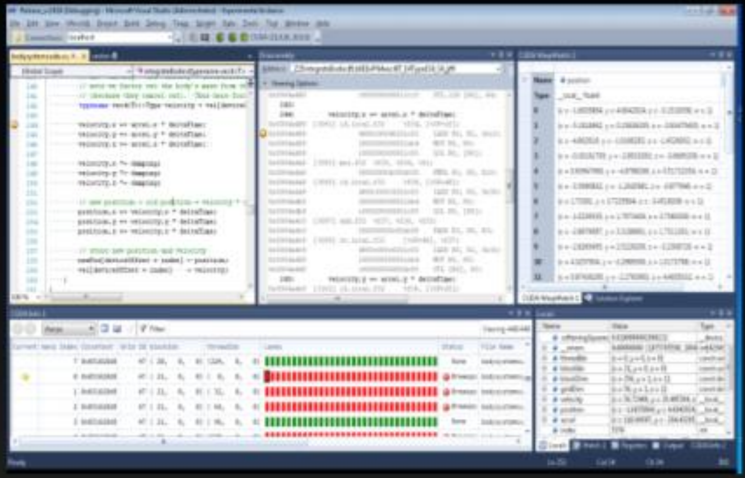
- Simultaneously debug CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

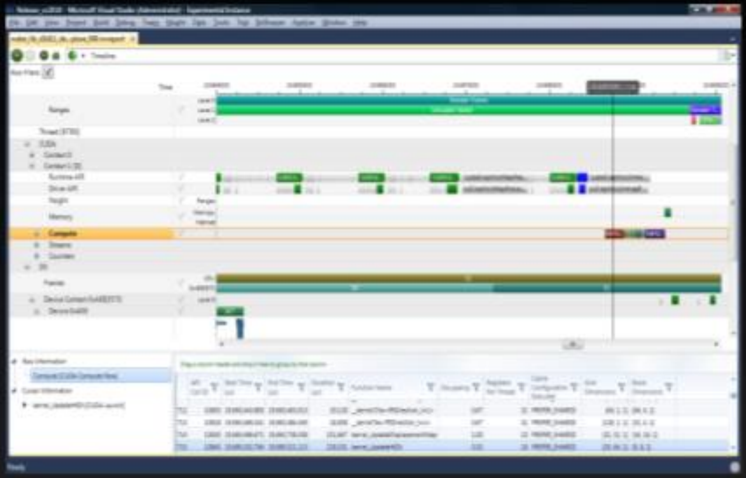
Nsight Profiler

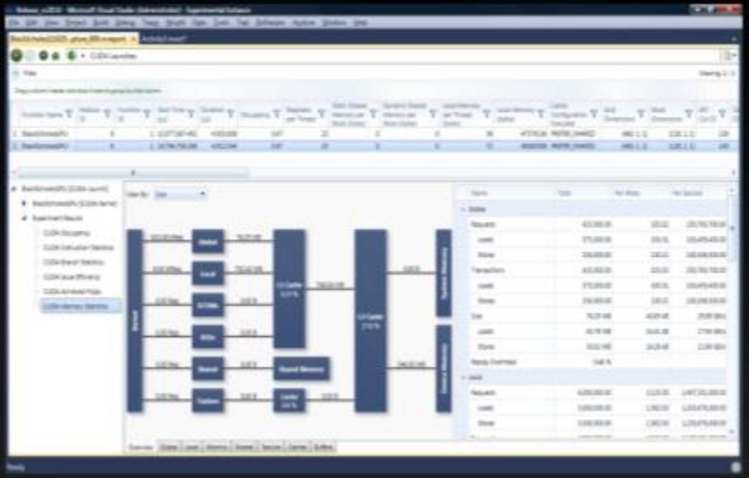
- Quickly identifies performance issues
- Integrated expert system
- Source line correlation

developer.nvidia.com/nsight

NVIDIA® Nsight™ Visual Studio Ed.







CUDA Debugger

- Debug CUDA kernels directly on GPU hardware
- Examine thousands of threads executing in parallel
- Use on-target conditional breakpoints to locate errors

CUDA Memory Checker

- Enables precise error detection

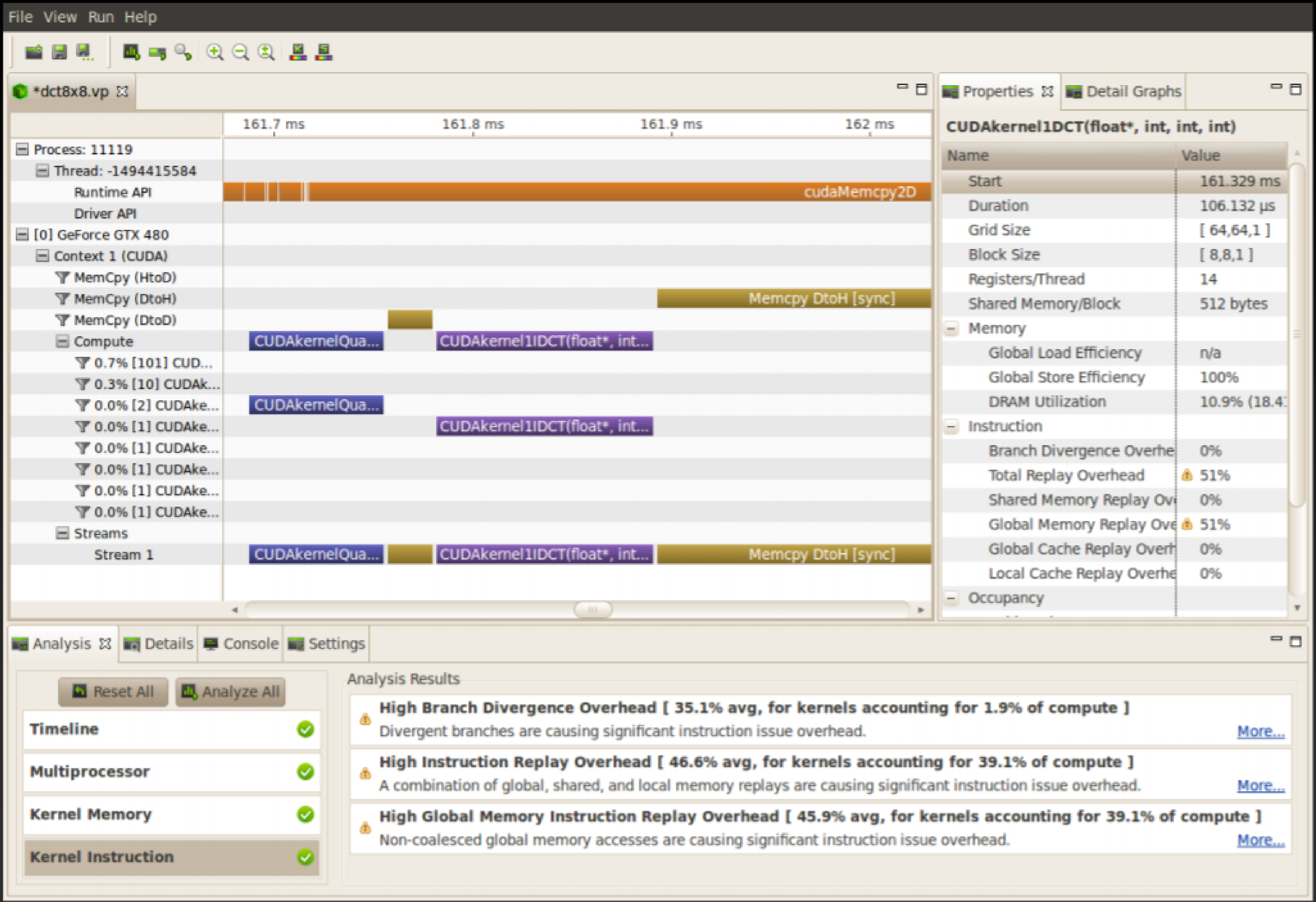
System Trace

- Review CUDA activities across CPU and GPU
- Perform deep kernel analysis to detect factors limiting maximum performance

CUDA Profiler

- Advanced experiments to measure memory utilization, instruction throughput and stalls

NVIDIA Visual Profiler



CUDAkernel1DCT(float*, int, int, int)

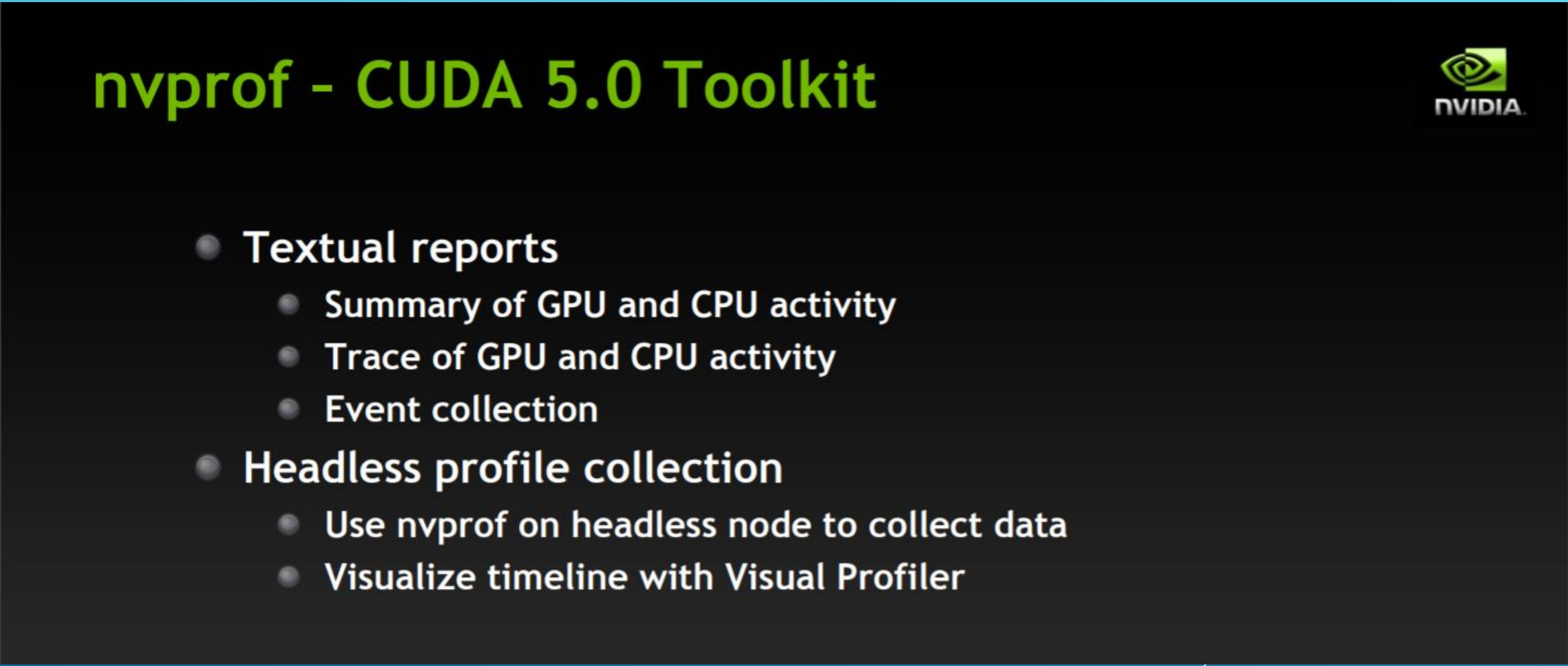
Name	Value
Start	161.329 ms
Duration	106.132 µs
Grid Size	[64,64,1]
Block Size	[8,8,1]
Registers/Thread	14
Shared Memory/Block	512 bytes
Memory	
Global Load Efficiency	n/a
Global Store Efficiency	100%
DRAM Utilization	10.9% (18.4
Instruction	
Branch Divergence Overhe	0%
Total Replay Overhead	⚠ 51%
Shared Memory Replay Ov	0%
Global Memory Replay Ove	⚠ 51%
Global Cache Replay Overh	0%
Local Cache Replay Overhe	0%
Occupancy	

Analysis Results

⚠ **High Branch Divergence Overhead [35.1% avg, for kernels accounting for 1.9% of compute]**
Divergent branches are causing significant instruction issue overhead. More...

⚠ **High Instruction Replay Overhead [46.6% avg, for kernels accounting for 39.1% of compute]**
A combination of global, shared, and local memory replays are causing significant instruction issue overhead. More...

⚠ **High Global Memory Instruction Replay Overhead [45.9% avg, for kernels accounting for 39.1% of compute]**
Non-coalesced global memory accesses are causing significant instruction issue overhead. More...



nvprof - CUDA 5.0 Toolkit

- Textual reports
  - Summary of GPU and CPU activity
  - Trace of GPU and CPU activity
  - Event collection
- Headless profile collection
  - Use nvprof on headless node to collect data
  - Visualize timeline with Visual Profiler

Links to get started

- Get CUDA: www.nvidia.com/getcuda
- Nsight: www.nvidia.com/nsight
- Programming Guide/Best Practices…
  - docs.nvidia.com
- Questions:
  - NVIDIA Developer forums devtalk.nvidia.com
  - Search or ask on www.stackoverflow.com/tags/cuda
- General: www.nvidia.com/cudazone

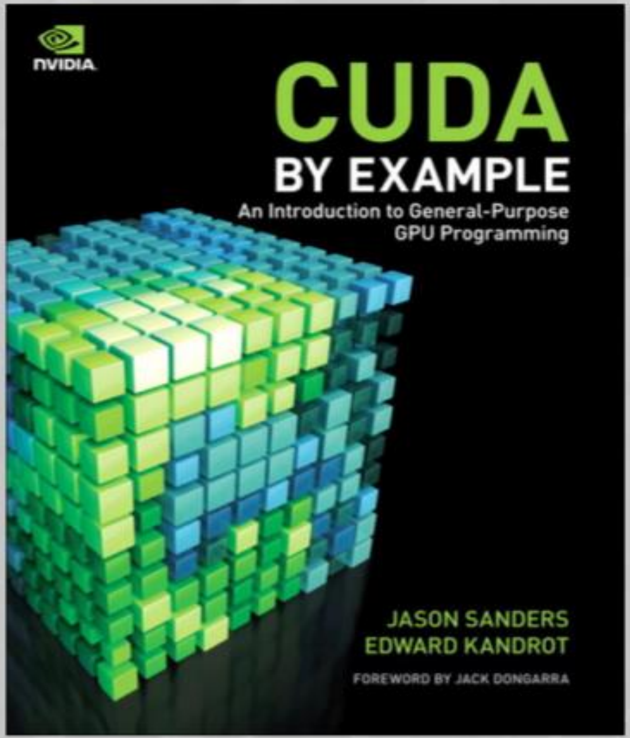
Developer Curriculum

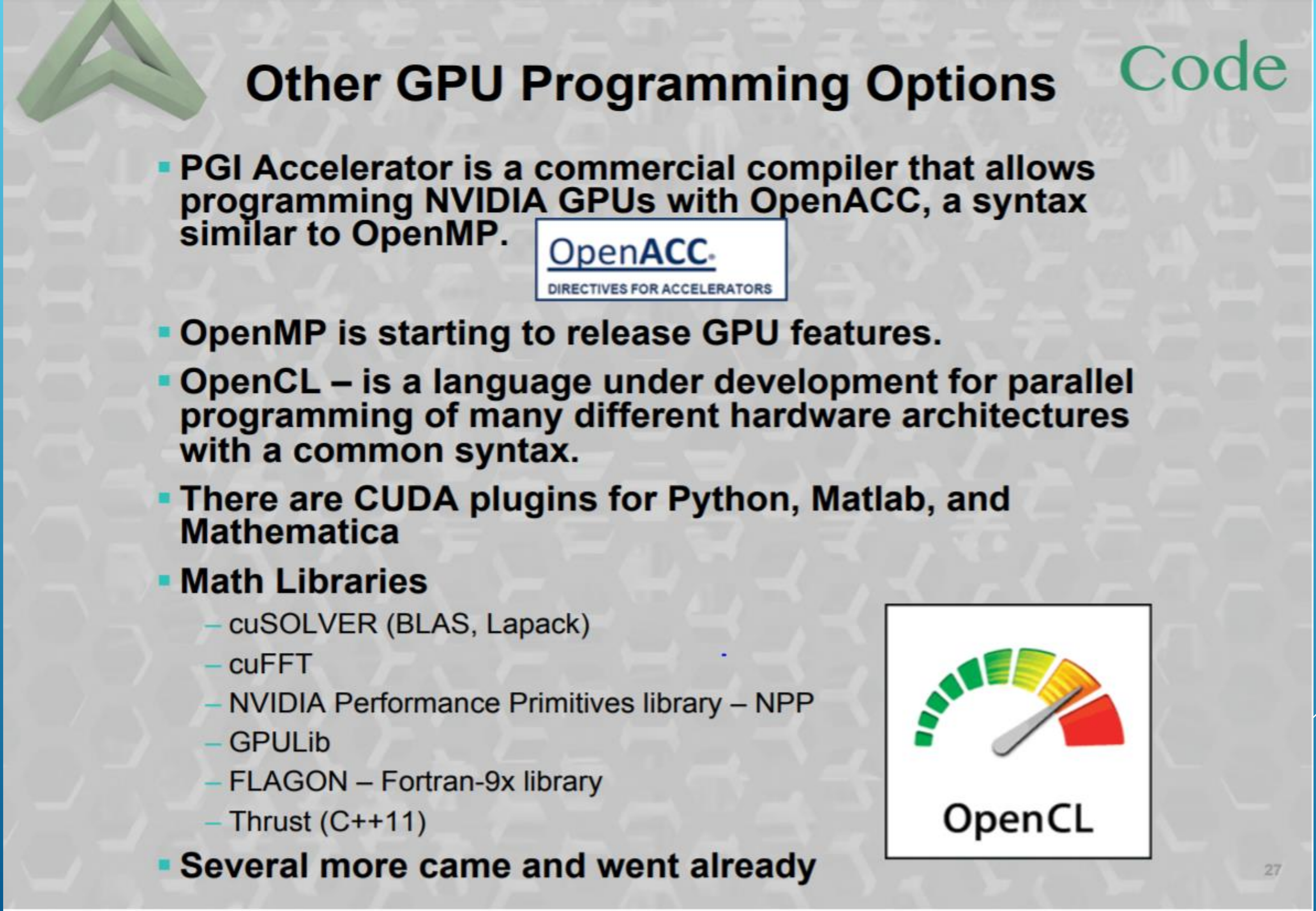
- Site: developer.nvidia.com/cuda-education
  - Mailing list is live!
- Forums for discussion:
  - Education section on: devtalk.nvidia.com

CUDA References

Doc

- On the Alabama Supercomputer Center systems, documentation is in the directory /opt/asn/doc/gpu
  - Start with README.txt and TIPS.txt
  - CUDA_C_Getting_Started_Linux.pdf
  - CUDA_C_Programming_Guide.pdf
  - CUDA_C_Best_Practices_Guide.pdf
  - Examples are in the portland_accelerator and portland_cuda_fortran directories
  - There is more information in the supplmental_docs directory
- A good introduction to CUDA programming
  - "CUDA BY EXAMPLE" by J. Sanders, E. Kandrot, Addison Wesley, 2011.


NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA



Other GPU Programming Options

Code

- PGI Accelerator is a commercial compiler that allows programming NVIDIA GPUs with OpenACC, a syntax similar to OpenMP.

OpenACC
DIRECTIVES FOR ACCELERATORS

- OpenMP is starting to release GPU features.
- OpenCL – is a language under development for parallel programming of many different hardware architectures with a common syntax.
- There are CUDA plugins for Python, Matlab, and Mathematica
- Math Libraries
  - cuSOLVER (BLAS, Lapack)
  - cuFFT
  - NVIDIA Performance Primitives library – NPP
  - GPULib
  - FLAGON – Fortran-9x library
  - Thrust (C++11)
- Several more came and went already

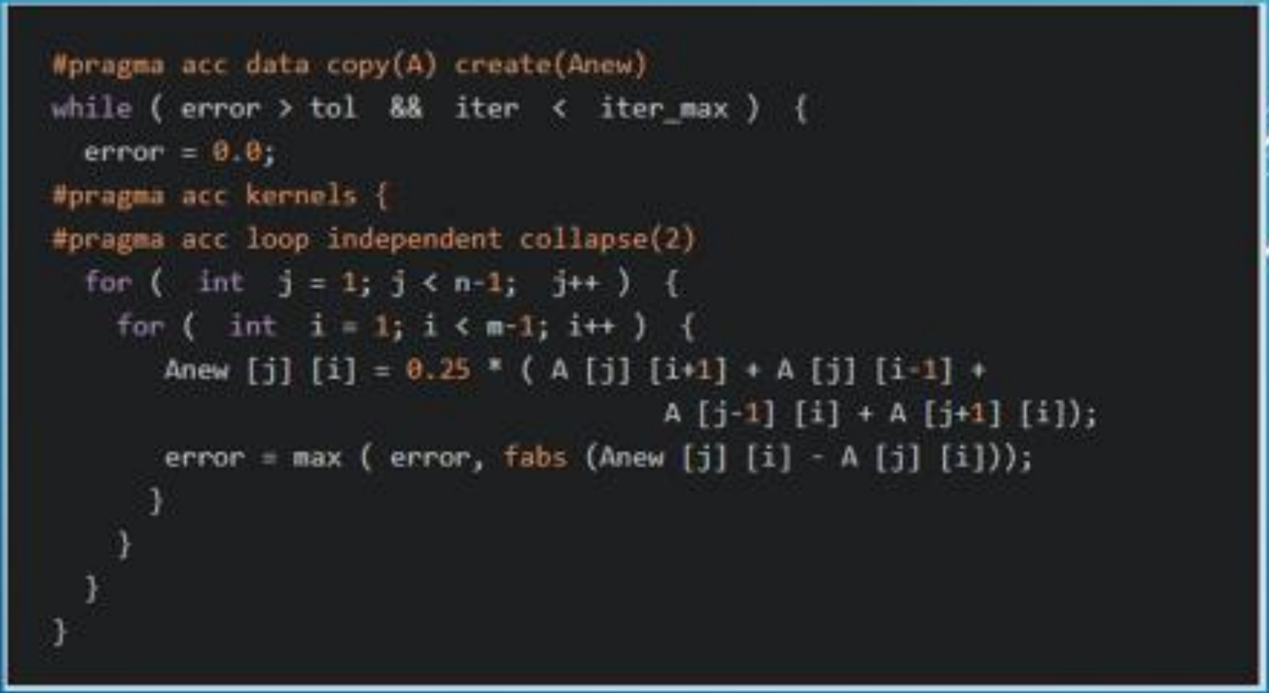
OpenCL

What is OpenACC?

OpenACC is a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.

Get Started or take the next steps

https://www.openacc.org/get-started

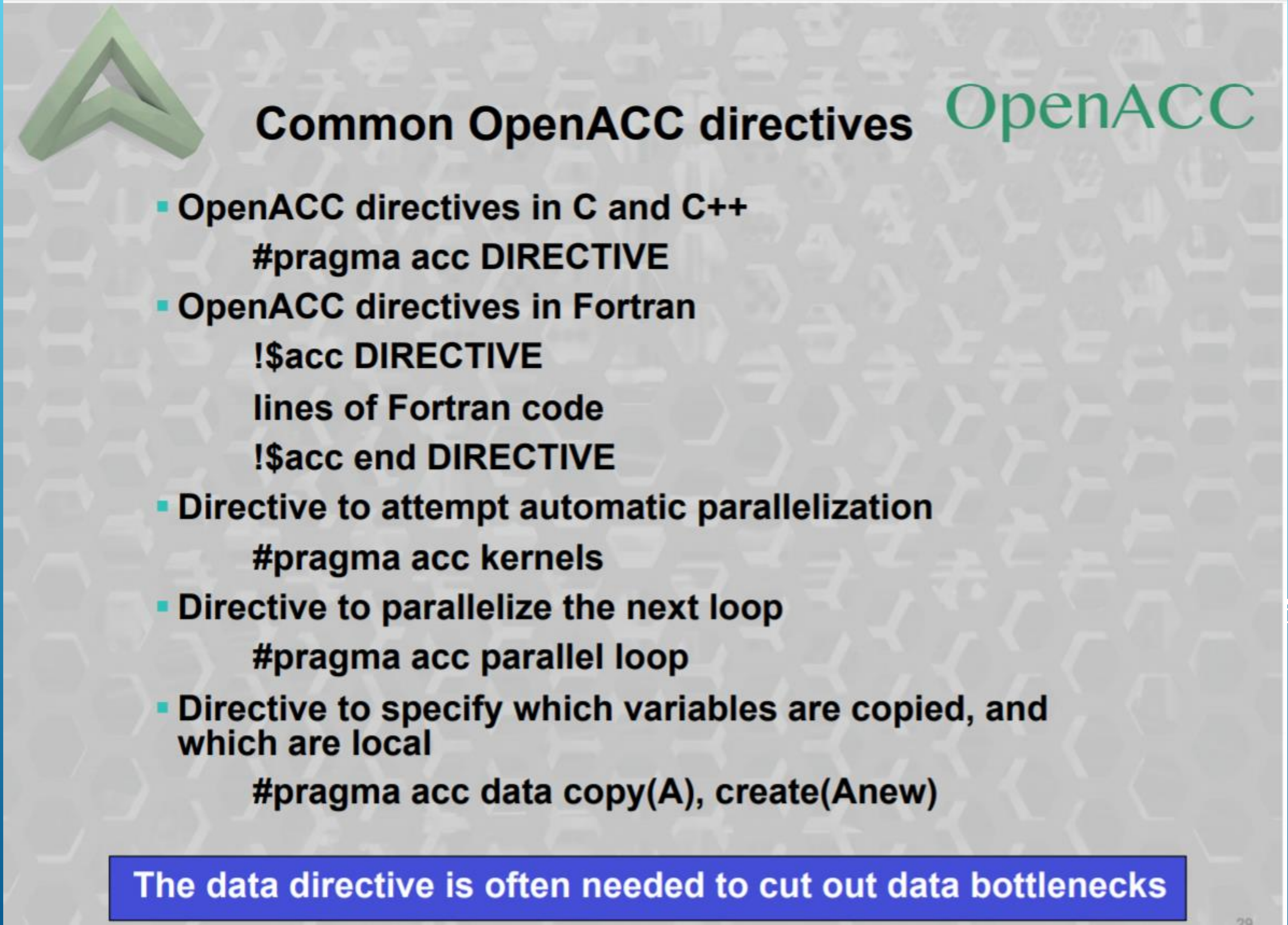


```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                            A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
  }
}
```

OpenACC Example

```
// OpenACC matrix add
int main() {
 int i, j;
#pragma acc kernels loop gang(32), vector (16)
 for (i=0;i<N;i++) {
#pragma acc loop gang(16), vector(32)
  for (j=0;j<N;j++) {
   C[i][j]=A[i][j]+B[i][j];
  }
 }
}
```

- openACC is easier to program than CUDA
- but less efficient, so the program wont run as fast



Common OpenACC directives

OpenACC

- OpenACC directives in C and C++

```
#pragma acc DIRECTIVE
```

- OpenACC directives in Fortran

```
!$acc DIRECTIVE
lines of Fortran code
!$acc end DIRECTIVE
```

- Directive to attempt automatic parallelization

```
#pragma acc kernels
```

- Directive to parallelize the next loop

```
#pragma acc parallel loop
```

- Directive to specify which variables are copied, and which are local

```
#pragma acc data copy(A), create(Anew)
```

The data directive is often needed to cut out data bottlenecks

Compiling and Running

OpenACC

- Typical compile command for C

```
pgcc -acc -Minfo=accel -ta=nvidia -o file file.c
```

- Environment variable to print GPU use information at run time

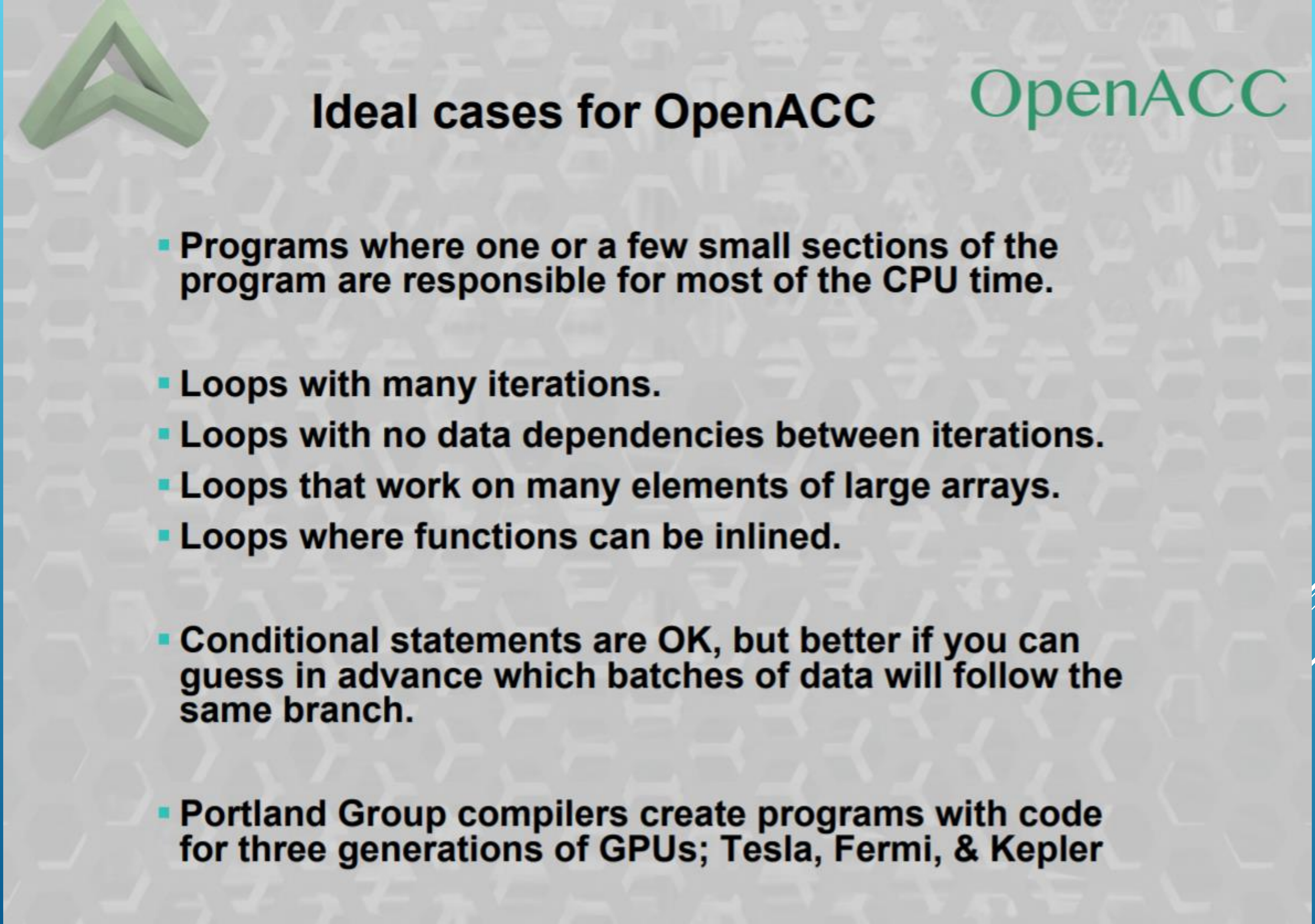
```
export PGI_ACC_TIME=1
```

- The program runs slightly slower with this turned on

- Environment variable to print out information about data transfers to the GPU at run time

```
export PGI_ACC_NOTIFY=3
```

- This slows down execution significantly



Ideal cases for OpenACC

OpenACC

- Programs where one or a few small sections of the program are responsible for most of the CPU time.

- Loops with many iterations.
- Loops with no data dependencies between iterations.
- Loops that work on many elements of large arrays.
- Loops where functions can be inlined.

- Conditional statements are OK, but better if you can guess in advance which batches of data will follow the same branch.

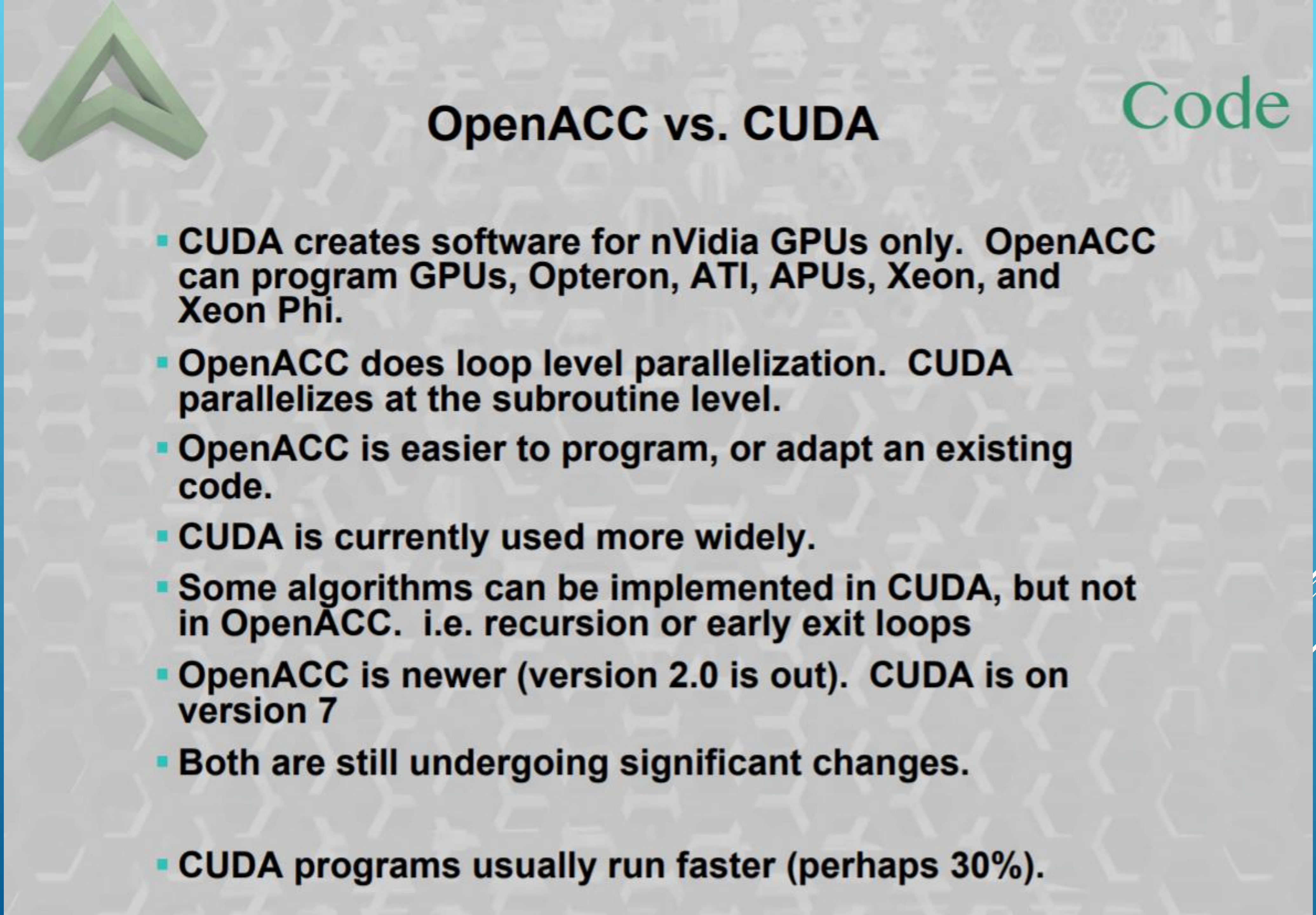
- Portland Group compilers create programs with code for three generations of GPUs; Tesla, Fermi, & Kepler

What Does NOT work well

OpenACC

- Loops with IO statements.
- Loops with early exits, including do-while loops.
- Loops with many branches to other functions.
- Pointer arithmetic

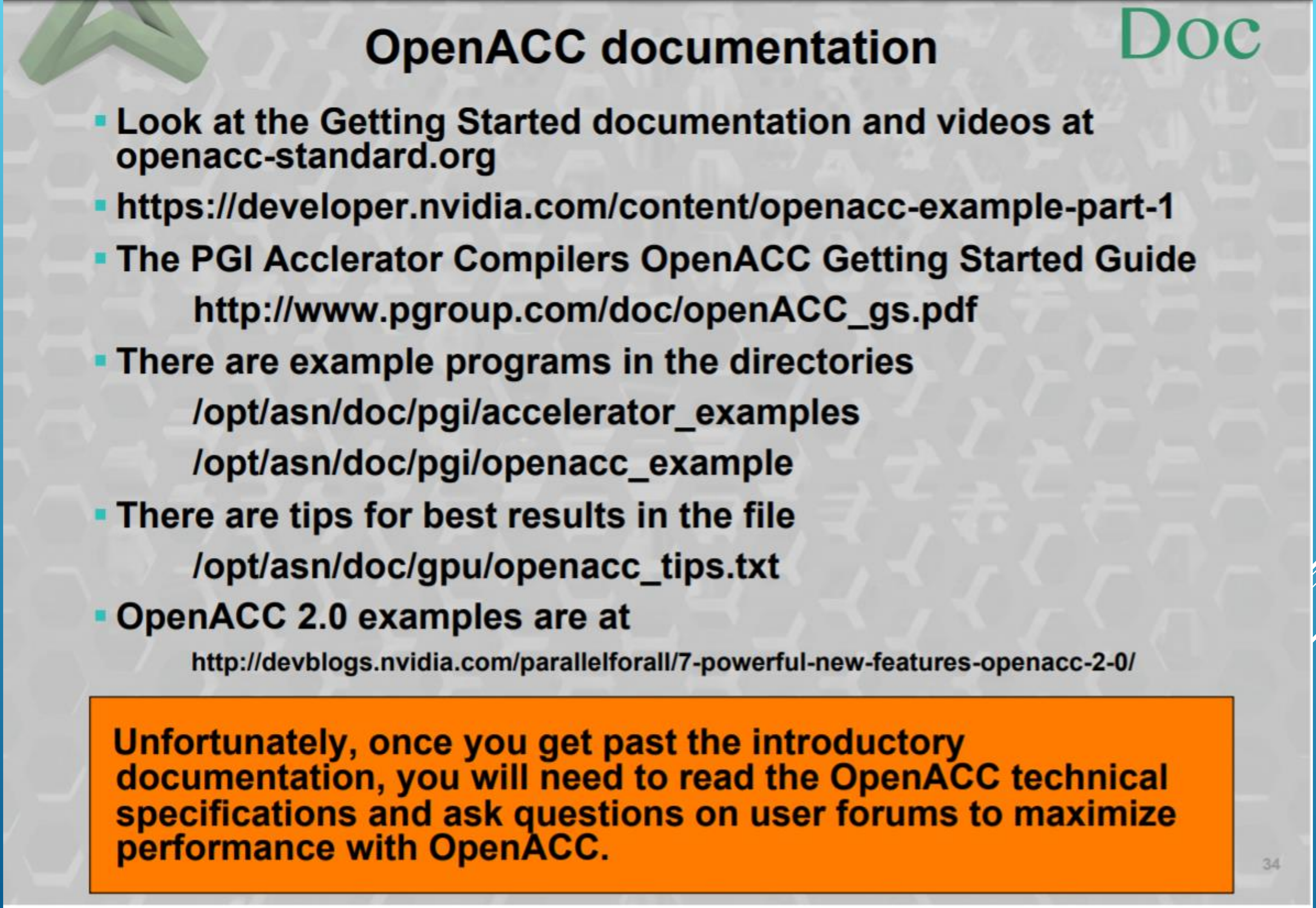
Confusingly, a failed compile creates a single processor executable.



Code

OpenACC vs. CUDA

- CUDA creates software for nVidia GPUs only. OpenACC can program GPUs, Opteron, ATI, APUs, Xeon, and Xeon Phi.
- OpenACC does loop level parallelization. CUDA parallelizes at the subroutine level.
- OpenACC is easier to program, or adapt an existing code.
- CUDA is currently used more widely.
- Some algorithms can be implemented in CUDA, but not in OpenACC. i.e. recursion or early exit loops
- OpenACC is newer (version 2.0 is out). CUDA is on version 7
- Both are still undergoing significant changes.
- CUDA programs usually run faster (perhaps 30%).



OpenACC documentation

- Look at the Getting Started documentation and videos at openacc-standard.org
- https://developer.nvidia.com/content/openacc-example-part-1
- The PGI Acclerator Compilers OpenACC Getting Started Guide

  http://www.pgroup.com/doc/openACC_gs.pdf
- There are example programs in the directories

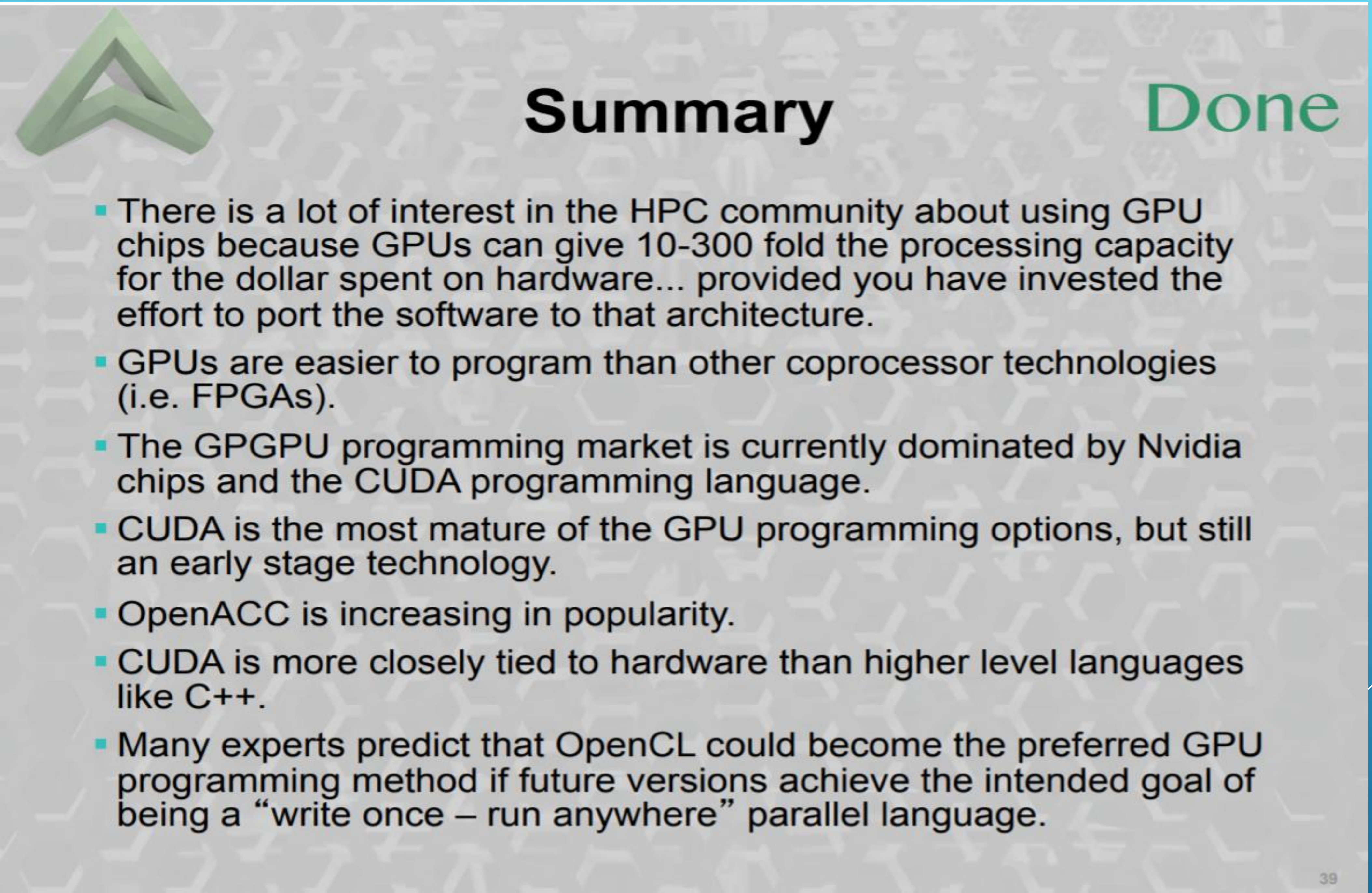
  /opt/asn/doc/pgi/accelerator_examples

  /opt/asn/doc/pgi/openacc_example
- There are tips for best results in the file

  /opt/asn/doc/gpu/openacc_tips.txt
- OpenACC 2.0 examples are at

  http://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/

Unfortunately, once you get past the introductory documentation, you will need to read the OpenACC technical specifications and ask questions on user forums to maximize performance with OpenACC.



Summary

Done

- There is a lot of interest in the HPC community about using GPU chips because GPUs can give 10-300 fold the processing capacity for the dollar spent on hardware... provided you have invested the effort to port the software to that architecture.
- GPUs are easier to program than other coprocessor technologies (i.e. FPGAs).
- The GPGPU programming market is currently dominated by Nvidia chips and the CUDA programming language.
- CUDA is the most mature of the GPU programming options, but still an early stage technology.
- OpenACC is increasing in popularity.
- CUDA is more closely tied to hardware than higher level languages like C++.
- Many experts predict that OpenCL could become the preferred GPU programming method if future versions achieve the intended goal of being a "write once – run anywhere" parallel language.