# The 2-day tutorial at a glance

<table>
<thead>
<tr>
<th>Day</th>
<th>Time frame</th>
<th>Contents</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>25/2</td>
<td>09:00 - 10:30</td>
<td>The GPU hardware: Many-core developments</td>
<td>Basic</td>
</tr>
<tr>
<td>25/2</td>
<td>10:30 - 11:00</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>25/2</td>
<td>11:00 - 12:30</td>
<td>CUDA Programming: Threads, blocks, kernels, grids</td>
<td>Basic</td>
</tr>
<tr>
<td>25/2</td>
<td>12:30 - 13:30</td>
<td>Lunch break</td>
<td></td>
</tr>
<tr>
<td>25/2</td>
<td>13:30 - 15:00</td>
<td>CUDA tools: Compiling, profiling, occupancy calculator</td>
<td>Basic</td>
</tr>
<tr>
<td>25/2</td>
<td>15:00 - 15:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>25/2</td>
<td>15:30 - 17:00</td>
<td>CUDA examples: VectorAdd, Stencils, ReverseArray, MxM</td>
<td>Basic</td>
</tr>
<tr>
<td>26/2</td>
<td>09:00 - 10:15</td>
<td>Inside Kepler &amp; Maxwell: Hyper-Q, dyn. par., unified memory</td>
<td>Advanced</td>
</tr>
<tr>
<td>26/2</td>
<td>10:15 - 10:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>26/2</td>
<td>10:30 - 11:30</td>
<td>OpenACC and other approaches to GPU computing</td>
<td>Intermediate</td>
</tr>
<tr>
<td>26/2</td>
<td>11:30 - 13:00</td>
<td>Lunch break</td>
<td></td>
</tr>
<tr>
<td>26/2</td>
<td>13:00 - 15:00</td>
<td>Hands-on (1): Programming GPUs</td>
<td>All</td>
</tr>
<tr>
<td>26/2</td>
<td>15:00 - 15:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>26/2</td>
<td>15:30 - 17:00</td>
<td>Hands-on (2): Programming GPUs</td>
<td>All</td>
</tr>
</tbody>
</table>
Tutorial contents for today [115 slides]

1. Introduction. [17 slides]
2. Architecture. [18]
   1. CUDA hardware model. [3]
   6. Síntesis generacional. [1]
4. Syntax. [16]
   1. Basic elements. [10]
5. Compilation and tools [12]
6. Examples: VectorAdd, Stencil, MxM. [25]
7. Bibliography, resources and tools. [12]
Prerequisites for this tutorial

You (probably) need experience with C.
You do not need parallel programming background (but it helps if you have it).
You do not need knowledge about the GPU architecture: We will start with the basic pillars.
You do not need graphics experience. Those were the old times (shaders, Cg). With CUDA, it is not required any knowledge about vertices, pixels, textures, ...
I. Introduction
Welcome to the GPU world

GeForce

Quadro

Tegra

Tesla
The characters of this story: The CUDA family picture

GPU Computing Applications

Libraries and Middleware
- CUFFT
- CUBLAS
- CURAND
- CUSPARSE
- CUDA
- MAGMA
- Thrust
- NPP
- VSIPL
- SVM
- OpenCurrent
- PhysX
- OptiX
- Iray
- MATLAB
- Mathematica

Programming Languages
- C
- C++
- Fortran
- Java
- Python
- Wrappers
- DirectCompute
- Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

- **Kepler Architecture** (compute capabilities 3.x)
  - GeForce 600 Series
  - Quadro Kepler Series
  - Tesla K20
  - Tesla K10

- **Fermi Architecture** (compute capabilities 2.x)
  - GeForce 500 Series
  - GeForce 400 Series
  - Quadro Fermi Series
  - Tesla 20 Series

- **Tesla Architecture** (compute capabilities 1.x)
  - GeForce 200 Series
  - GeForce 9 Series
  - GeForce 8 Series
  - Quadro FX Series
  - Quadro Plex Series
  - Quadro NVS Series
  - Tesla 10 Series
The impressive evolution of CUDA

**Year 2008**
- **100.000.000** CUDA-capable GPUs
- **150.000** CUDA downloads
- **1** supercomputer
- **60** university courses
- **4.000** academic papers

**Year 2015**
- **600.000.000** CUDA-capable GPUs
- **5.000.000** CUDA downloads
- **75** supercomputers
- **840** university courses
- **50.000** academic papers

The CUDA software is downloaded more than once a minute.

Manuel Ujaldon - Nvidia CUDA Fellow
Worldwide distribution of CUDA university courses
Summary of GPU evolution

2001: First many-cores (vertex and pixel processors).
2003: Those processor become programmable (with Cg).
2006: Vertex and pixel processors unify.
2007: CUDA emerges.
2008: Double precision floating-point arithmetic.
2010: Operands are IEEE-normalized and memory is ECC.
2012: Wider support for irregular computing.
2014: The CPU-GPU memory space is unified.
Still pending: Reliability in clusters and connection to disk.
The 3 features which have made the GPU such a unique processor

- **Simplified.**
  - The control required for one thread is amortized by 31 more (warp).

- **Scalability.**
  - Makes use of the huge data volume handled by applications to define a sustainable parallelization model.

- **Productivity.**
  - Endowed with efficient mechanisms for switching immediately to another thread whenever the one being executed suffers from stalls.

**CUDA essential keywords:**
- Warp, SIMD, latency hiding, free context switch.
What is CUDA?
“Compute Unified Device Architecture”

A platform designed jointly at software and hardware levels to make use of the GPU computational power in general-purpose applications at three levels:

- **Software**: It allows to program the GPU with minimal but powerful SIMD extensions to enable heterogeneous programming and attain an efficient and scalable execution.

- **Firmware**: It offers a driver oriented to GPGPU programming, which is compatible with that used for rendering. Straightforward APIs to manage devices, memory, etc.

- **Hardware**: It exposes GPU parallelism for general-purpose computing via a number of multiprocessors endowed with cores and a memory hierarchy.
CUDA C at a glance

Essentially, it is C language with minimal extensions:
- Programmer writes the program for a single thread, and the code is automatically instanciated over hundreds of threads.

CUDA defines:
- An architectural model:
  - With many processing cores grouped in multiprocessors who share a SIMD control unit.
- A programming model:
  - Based on massive data parallelism and fine-grained parallelism.
  - Scalable: The code is executed on a different number of cores without recompiling it.
- A memory management model:
  - More explicit to the programmer, where caches are not transparent anymore.

Goals:
- Build a code which scales to hundreds of cores in a simple way, allowing us to declare thousands of threads.
- Allow heterogeneous computing (between CPUs and GPUs).
Terminology:

Host: The CPU and the memory on motherboard [DDR3 as of 2013].

Device: The graphics card [GPU + video memory]:

- GPU: Nvidia GeForce/Tesla.
- Video memory: GDDR5 as of 2015.
CUDA executes a program on a device (the GPU), which is seen as a co-processor for the host (the CPU).

CUDA can be seen as a library of functions which contains 3 types of components:

- **Host**: Control and access to devices.
- **Device**: Specific functions for the devices.
- **All**: Vector data types and a set of routines supported on both sides.
The code to be written in CUDA can be lower than 5%, but exceed 50% of the execution time if remains on CPU.
#include <iostream>
#include <algorithm>
using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__
void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}

HOST CODE:
- Serial code.
- Parallel code.
- Serial code.

DEVICE CODE:
Parallel function written in CUDA.
Simple Processing Flow (1/3)

1. Copy input data from CPU memory to GPU memory.
1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.
Simple Processing Flow (3/3)

1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.
3. Transfer results from GPU memory to CPU memory.
The classic example

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Salida:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- Standard C that runs on the host.
- NVIDIA compiler (nvcc) can be used to compile programs with no device code.
Hello World! with device code (1/2)

```c
__global__ void mykernel(void)
{
}

int main(void)
{
    mykernel<<<1,1>>>()
    printf("Hello World!\n");
    return 0;
}
```

Two new syntactic elements:
- The CUDA C keyword `__global__` indicates a function that runs on the device and is called from host code.
- `mykernel<<<1,1>>>` is a CUDA kernel launch from the host code.

That's all that is required to execute a function on the GPU!

nvcc separates source code into host and device.
- Device functions (like `mykernel()`) are processed by NVIDIA compiler.
- Host functions (like `main()`) are processed by host compiler (`gcc` for Unix, `cl.exe` for Windows).

Manuel Ujaldon - Nvidia CUDA Fellow
Hello World! with device code (2/2)

```c
__global__ void mykernel(void)
{
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- mykernel() does nothing this time.
- Triple angle brackets mark a call from host code to device code.
  - Also called a “kernel launch”.
  - Parameters `<<<1,1>>>` describe CUDA parallelism (blocks and threads).
II. Architecture
II.1. CUDA hardware model
Overview of CUDA hardware generations

- Tesla (2008)
  - CUDA
  - FP64

- Fermi (2010)
  - Unified memory
  - Dynamic Parallelism

- Kepler (2012)
  - 3D Memory
  - NVLink

- Maxwell (2014)
  - Unified memory
  - DX12

- Pascal (2016)
  - 3D Memory
  - NVLink

GFLOPS in double precision for each watt consumed

- 2008: 2
- 2010: 6
- 2012: 14
- 2014: 22
- 2016: 24

Manuel Ujaldon - Nvidia CUDA Fellow
The CUDA hardware model: SIMD processors structured, a tale of hardware scalability

A GPU consists of:
- N multiprocessors (or SMs), each containing M cores (or stream procs).

Massive parallelism:
- Applied to thousands of threads.
- Sharing data at different levels.

Heterogeneous computing:
- GPU:
  - Data intensive.
  - Fine-grain parallelism.
- CPU:
  - Control/management.
  - Coarse-grain parallelism.

<table>
<thead>
<tr>
<th></th>
<th>G80 (Tesla)</th>
<th>GT200 (Tesla)</th>
<th>GF100 (Fermi)</th>
<th>GK110 (Kepler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time period</td>
<td>2006-07</td>
<td>2008-09</td>
<td>2010-11</td>
<td>2012-13</td>
</tr>
<tr>
<td>N (multiprocs.)</td>
<td>16</td>
<td>30</td>
<td>14-16</td>
<td>13-15</td>
</tr>
<tr>
<td>M (cores/multip.)</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>192</td>
</tr>
<tr>
<td>Number of cores</td>
<td>128</td>
<td>240</td>
<td>448-512</td>
<td>2496-2880</td>
</tr>
</tbody>
</table>
Memory hierarchy

Each multiprocessor has:
- A register file.
- Shared memory.
- A constant cache and a texture cache, both read-only.

Global memory is the actual video memory (GDDR5):
- Three times faster than the DDR3 used by the CPU, but...
- ... around 500 times slower than shared memory! (DRAM versus SRAM).
II.2. The first generation: Tesla (G80 and GT200)
The first generation: G80 (GeForce 8800)

GPU G80 (around 600 MHz, much lower than the frequency for the cores)

- Multiprocessor 16
  - Multiprocessor 2
    - Multiprocessor 1 (CUDA thread-blocks are mapped onto multiprocessors)
      - Shared memory (16 KB)
      - Registers
      - Core 1 (1.35 GHz)
      - Core 2
      - Texture cache
      - Registers
      - Core 8
      - Control Unit (issues SIMD instructions)

- Global memory (up to 1.5 GB) (GDDR3 @ 2x 800 MHz)
The first generation: GT200 (GTX 200)

GPU GTX 200 (around 600 MHz)

Multiprocessor 30

Multiprocessor 2

Multiprocessor 1 (CUDA thread-blocks are mapped onto multiprocessors)

Shared memory (16 KB)

Registers

Core 1 (1.30 GHz)

Core 2

…”

Core 8

Control Unit (issues SIMD instructions)

(CUDA kernels are mapped to GPU cores)

Global memory (up to 4 GB) (GDDR3, 512 bits @ 2x 1.1GHz = 141.7 GB/s)
Scalability for future generations: Alternatives for increasing performance

- Raise the number of multiprocessors (basic node), that is, we grow over the Z dimension. This is the path followed by 1st gener. (16 to 30).
- Raise the number of processors within a multiprocessor, which means growing over the X dimension. That is what the 2nd and 3rd gener. have done (from 8 to 32 and from there to 192).
- Increment the size of shared memory (extending the Y dim.).
II. 3. The second generation: Fermi (GF1xx)
# Fermi hardware compared to its predecessors

<table>
<thead>
<tr>
<th>GPU architecture</th>
<th>G80</th>
<th>GT200</th>
<th>GF100 (Fermi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial sample</td>
<td>GeForce 8800</td>
<td>GTX 200</td>
<td>GTX 580</td>
</tr>
<tr>
<td>Year released</td>
<td>2006</td>
<td>2008</td>
<td>2010</td>
</tr>
<tr>
<td>Number of transistors</td>
<td>681 millions</td>
<td>1400 millions</td>
<td>3000 millions</td>
</tr>
<tr>
<td>Integer and fp32 cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>fp64 (double precision)</td>
<td>0</td>
<td>30</td>
<td>256</td>
</tr>
<tr>
<td>Double precision floating-point speed</td>
<td>None</td>
<td>30 madds/cycle</td>
<td>256 madds/cycle</td>
</tr>
<tr>
<td>Warp scheduler(s)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared memory size</td>
<td>16 KB</td>
<td>16 KB</td>
<td>16 KB + 48 KB (or vice versa)</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>None</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>L2 cache size</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>DRAM error correction</td>
<td>No</td>
<td>No</td>
<td>Yes (elective)</td>
</tr>
<tr>
<td>Address bus (width)</td>
<td>32 bits</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
</tbody>
</table>
Fermi: An architectural overview

- Up to 512 cores (16 SMs, each endowed with 32 cores).
- Dual scheduler at the front-end of each SM.
- 64 KB on each SM for shared memory and L1 cache.
Fermi is the first GPU with a L1 cache, combined with shared memory for a total of 64 KB for each SM (32 cores). 64 KB are split into 3:1 or 1:3 proportions (programmer’s choice).

There is also a L2 cache of 768 KB, with data conherence shared by all multiprocessors (SMs).
II. 4. The third generation: Kepler (GK1xx)
Kepler GK110 Block Diagram

- 7.1 billion transistors.
- 15 SMX multiprocs.
- > 1 TFLOP FP64.
- 1.5 MB L2 Cache.
- 384-bit GDDR5.
- PCI Express Gen3.
Multiprocessor evolution:
From SMs in Fermi to SMXs in Kepler
The SMX multiprocessor

Instruction scheduling and issuing in **warps**

Instructions execution.
512 functional units:
- 192 for ALUs.
- 192 for FPUs S.P.
- 64 for FPUs D.P.
- 32 for load/store.
- 32 for SFUs (log, sqrt, ...)

Memory access

Front-end

Back-end

Interface
II. 5. The fourth generation: Maxwell (GM1xx)
Maxwell and SMM multiprocessors (for GeForce GTX 750 Ti, GM107 with 5 SMM)

- 1870 Mt.
- 148 mm².
The SMMs

- Keep the same 4 warp schedulers, and the same LD/ST and SFU units.
- Reduce the number of cores for int and float: from 192 to 128 units.
## Some commercial models on 28 nm.

<table>
<thead>
<tr>
<th></th>
<th>GeForce GTX 650</th>
<th>GeForce GTX 650 Ti</th>
<th>GeForce GTX 750 Ti</th>
<th>GeForce GTX 660</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>GK107</td>
<td>GK106</td>
<td>GM107</td>
<td>GK106</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Kepler</td>
<td>Kepler</td>
<td>Maxwell</td>
<td>Kepler</td>
</tr>
<tr>
<td><strong>Multiprocessors</strong></td>
<td>2 SMX</td>
<td>4 SMX</td>
<td>5 SMM</td>
<td>5 SMX</td>
</tr>
<tr>
<td><strong>Number of cores</strong></td>
<td>192 x 2 = 384</td>
<td>192 x 4 = 768</td>
<td>128 x 5 = 640</td>
<td>192 x 5 = 960</td>
</tr>
<tr>
<td><strong>Frequency of cores</strong></td>
<td>1058 MHz</td>
<td>925 MHz</td>
<td>1020 - <strong>1085 MHz</strong></td>
<td>980 - 1033 MHz</td>
</tr>
<tr>
<td><strong>DRAM bus width</strong></td>
<td>128 bits</td>
<td>128 bits</td>
<td>128 bits</td>
<td>192 bits</td>
</tr>
<tr>
<td><strong>DRAM frequency</strong></td>
<td>2x 2500 MHz</td>
<td>2x 2700 MHz</td>
<td>2x 2700 MHz</td>
<td>2x 3000 MHz</td>
</tr>
<tr>
<td><strong>RAM bandwidth</strong></td>
<td>80 Gbytes/s.</td>
<td>86.4 Gbytes/s.</td>
<td>86.4 Gbytes/s.</td>
<td>144 Gbytes/s.</td>
</tr>
<tr>
<td><strong>GDDR5 memory size</strong></td>
<td>1 or 2 Gbytes</td>
<td>1 or 2 Gbytes</td>
<td>1 or 2 Gbytes</td>
<td>2 Gbytes</td>
</tr>
<tr>
<td><strong>Power connector</strong></td>
<td>6 pins</td>
<td>6 pins</td>
<td><strong>None</strong></td>
<td>6 pins</td>
</tr>
<tr>
<td><strong>Maximum TDP</strong></td>
<td>64 W.</td>
<td>110 W.</td>
<td><strong>60 W.</strong></td>
<td>140 W.</td>
</tr>
<tr>
<td><strong>Million transistors</strong></td>
<td>1300</td>
<td>2540</td>
<td>1870</td>
<td>3175 mm² ?</td>
</tr>
<tr>
<td><strong>Die size</strong></td>
<td>118 mm²</td>
<td>221 mm²</td>
<td>148 mm²</td>
<td>275 mm² ?</td>
</tr>
<tr>
<td><strong>Approx. cost (2 GB)</strong></td>
<td>100 €</td>
<td>110 €</td>
<td>110 €</td>
<td>150 €</td>
</tr>
</tbody>
</table>

Manuel Ujaldon - Nvidia CUDA Fellow
Major enhancements

KEPLER

MAXWELL 1st Generation

135% Performance/Core

2x Performance/Watt
Power efficiency

- GTX 750 Ti (1st Gen Maxwell)
- GTX 650 Ti (Kepler)
- GTX 550 Ti (Fermi)

Comparison over years:
- 2010: GTX 550 Ti (Fermi)
- 2012: GTX 650 Ti (Kepler)
- 2014: GTX 750 Ti (1st Gen Maxwell)
II. 6. A summary of four generations
## Scalability for the architecture: A summary of four generations

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Compute Capability</td>
<td>1.0</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td>N (multiprocs.)</td>
<td>16</td>
<td>30</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>M (cores/multip.)</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Number of cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
<td>336</td>
</tr>
</tbody>
</table>
III. Programming
Comparing the GPU and the CPU
From POSIX threads in CPU to CUDA threads in GPU

<table>
<thead>
<tr>
<th>POSIX-threads in CPU</th>
<th>CUDA in GPU, followed by host code in CPU</th>
<th>2D configuration: Grid of 2x2 blocks, 4 threads each</th>
</tr>
</thead>
</table>
| #define NUM_THREADS 16  void *myfun (void *threadId)  
{  
  int tid = (int) threadId;
  float result = sin(tid) * tan(tid);
  pthread_exit(NULL);
} |
| #define NUM_BLOCKS 1  
#define BLOCKSIZE 16  
__global__ void mykernel()  
{  
  int tid = threadIdx.x;
  float result = sin(tid) * tan(tid);
}  
void main()  
{  
  int t;  
  for (t=0; t<NUM_THREADS; t++)  
    pthread_create(NULL,NULL,myfun,t);  
  pthread_exit(NULL);
} |
| #define NUM_BLOCKS 1  
#define BLOCKSIZE 16  
__global__ void mykernel()  
{  
  int tid = threadIdx.x;  
  float result = sin(tid) * tan(tid);
}  
void main()  
{  
  int t;  
  for (t=0; t<NUM_THREADS; t++)  
    pthread_create(NULL,NULL,myfun,t);  
  pthread_exit(NULL);
} |
| #define NUM_BLOCKS 1  
#define BLOCKSIZE 16  
__global__ void mykernel()  
{  
  int tid = threadIdx.x;  
  float result = sin(tid) * tan(tid);
}  
void main()  
{  
  int t;  
  for (t=0; t<NUM_THREADS; t++)  
    pthread_create(NULL,NULL,myfun,t);  
  pthread_exit(NULL);
} |
The CUDA programming model

The GPU (device) is a highly multithreaded coprocessor to the CPU (host):

- Has its own DRAM (device memory).
- Executes many threads in parallel on several multiprocessor cores.

CUDA threads are extremely lightweight.

- Very little creation overhead.
- Context switching is essentially free.

Programmer’s goal: Declare thousands of threads to ensure the full utilization of hardware resources.
The model instanciates over few features to produce the commercial catalog

We may expect higher differences for these features between models associated to different generations.

Differences will also grow when graphics cards aim to different ends:

- $300-500 high-end graphics card.
- $150-250 mid-end.
- $60-120 low-end.

Video memory may also differ when a new technology emerges. Last step forward: GDDR5 vs. GDDR3.

Graphical features are different too, but that is out of the scope of this tutorial.
Each multiprocessor (SM) processes batches of blocks one after another.

- Active blocks = blocks processed by one multiprocessor in one batch.
- Active threads = all the threads from the active blocks.

Registers and shared memory within a multiprocessor are split among the active threads. Therefore, for any given kernel, the number of active blocks depends on:
  - The number of registers that the kernel requires.
  - How much shared memory the kernel consumes.
Programmers face the challenge of exposing parallelism for thousands cores using the following elements:

- **Device** = GPU = Set of multiprocessors.
- **Multiprocessor** = Set of processors + shared memory.
- **Kernel** = Program ready to run on GPU.
- **Grid** = Array of thread blocks that execute a kernel.
- **Thread block** = Group of SIMD threads that:
  - Execute a kernel on different data based on threadID and blockID.
  - Can communicate via shared memory.
- **Warp size** = 32. This is the granularity of the scheduler for issuing threads to the execution units.
Mapping hardware and software concepts

GPU
- Multiprocessor N
- Multiprocessor 2
- Multiprocessor 1

Shared memory
- Registers
- Processor 1
- Processor 2
- Processor M

Control Unit (SIMD)

Constant cache

Texture cache

Global memory

Thread
- per-Thread Private Local Memory

Thread Block
- per-Block Shared Memory

Grid 0
- per-Application Context
- Global Memory

Grid 1

Manuel Ujaldon - Nvidia CUDA Fellow
# Resources and limitations depending on CUDA hardware generation (CCC)

<table>
<thead>
<tr>
<th></th>
<th>CUDA Compute Capability (CCC)</th>
<th>Limitation</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0, 1.1</td>
<td>1.2, 1.3</td>
<td>2.0, 2.1</td>
</tr>
<tr>
<td>Multiprocessors / GPU</td>
<td>16</td>
<td>30</td>
<td>14-16</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
<td></td>
<td>Scalability</td>
</tr>
<tr>
<td>Cores / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
<td></td>
<td>Scalability</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Software</td>
<td></td>
<td>Throughput</td>
</tr>
<tr>
<td>Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Software</td>
<td></td>
<td>Throughput</td>
</tr>
<tr>
<td>Threads / Block</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td>Software</td>
<td></td>
<td>Parallelism</td>
</tr>
<tr>
<td>Threads / Multiprocessor</td>
<td>768</td>
<td>1 024</td>
<td>1 536</td>
</tr>
<tr>
<td></td>
<td>Software</td>
<td></td>
<td>Parallelism</td>
</tr>
<tr>
<td>32 bits registers / Multip.</td>
<td>8K</td>
<td>16K</td>
<td>32K</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
<td></td>
<td>Working set</td>
</tr>
<tr>
<td>Shared memory / Multip.</td>
<td>16K</td>
<td>16K</td>
<td>16K, 48K</td>
</tr>
<tr>
<td></td>
<td>Hardware</td>
<td></td>
<td>Working set</td>
</tr>
</tbody>
</table>
Threads are assigned to multiprocessors in blocks, and to cores via warps, which is the scheduling unit (32 threads). Threads of a block share information via shared memory, and can synchronize via `syncthreads()` calls.
Playing with parallel constraints in Kepler to maximize concurrency

- 1 block of 2048 threads. Forbidden by [2].
- 2 blocks of 1024 threads. Feasible on the same multiproc.
- 4 blocks of 512 threads. Feasible on the same multiproc.
- 8 blocks of 256 threads. Feasible on the same multiproc.
- 256 blocks of 8 threads. Forbidden by [1] on the same multiprocessor, feasible involving 16 multiprocessors.
Threads within a block can use the shared memory to perform tasks in a more cooperative and faster manner.

Global memory is the only visible to threads, blocks and kernels.
Playing with memory constraints in Kepler to maximize the use of resources

- Limits within a multiprocessor (SMX): 64 Kregs. and 48 KB. of shared memory. That way:
  - To allow a second block to execute on the same multiprocessor, each block must use at most 32 Kregs. and 24 KB of shared memory.
  - To allow a third block to execute on the same multiprocessor, each block must use at most 21.3 Kregs. and 16 KB. of shared mem.
- ... and so on. In general, the less memory used, the more concurrency for blocks execution.
- There is a trade-off between memory and parallelism!
Think small: 1D partitioning on a 64 elements vector

Remember: Use finest grained parallelism (assign one data to each thread). Threads and blocks deployment:

- 8 blocks of 8 threads each. Risk on smaller blocks: Waste parallelism if the limit of 8-16 blocks per multip. is reached.

- 4 blocks of 16 threads each. Risk on larger blocks: Squeeze the working set for each thread (remember that shared memory and register file are shared by all threads).
Now think big:
1D partitioning on a 64 million elems. array

- Maximum number of threads per block:
  - 1K on Fermi and Kepler.

- Maximum number of blocks:
  - 64K on Fermi, 4G on Kepler.

Larger sizes for data structures can only be covered with a huge number of blocks (keeping fine-grained parallelism).

- Choices:
  - 64K blocks of 1K threads each.
  - 128K blocks of 512 threads each (only feasible in Kepler).
  - 256K blocks of 256 threads each (only feasible in Kepler).
  - ... and so on.
Memory spaces

The CPU and the GPU have separated memory spaces:

- To communicate them, we use the PCI express bus.
- The GPU uses specific functions to allocate memory and copy data from CPU in a similar manner to what we are used with the C language (`malloc/free`).

Pointers are only addresses:

- You cannot derive from a pointer value if the address belongs to either the CPU or the GPU space.
- You have to be very careful when handling pointers, as the program usually crashes when a CPU data attempt to be accessed from GPU and vice versa (this situation is changing in CUDA 5.0, where the memory accessed from both processors is unified).
IV. Syntax
IV. 1. Basic elements
CUDA is C with some extra keywords. A preliminar example

```c
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke the SAXPY function sequentially
saxpy_serial(n, 2.0, x, y);
```

C code on the CPU

Equivalent CUDA code for its parallel execution on GPUs:

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    // More on parallel access patterns later in example 2
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke SAXPY in parallel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```
List of extensions added to the C language

Type qualifiers:
- global, device, shared, local, constant.

Keywords:
- threadIdx, blockIdx, gridDim, blockDim.

Intrinsics:
- __syncthreads();

Runtime API:
- Memory, symbols, execution management.

Kernel functions to launch code to the GPU from the CPU.

```c
__device__ float array[N];
__global__ void med_filter(float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx.x] = image[i];

    __syncthreads();
    ...
    image[j] = result;
}

// Allocate memory in the GPU
void *myimage;
cudaMalloc(&myimage, bytes);

// 100 thread blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```
Interaction between CPU and GPU

CUDA extends the C language with a new type of function, kernel, which executes code in parallel on all active threads within GPU. Remaining code is native C executed on CPU.

The typical main() of C combines the sequential execution on CPU and the parallel execution on GPU of CUDA kernels.

A kernel is launched in an asynchronous way, that is, control always returns immediately to the CPU.

Each GPU kernel has an implicit barrier when it ends, that is, it does not conclude until all its threads are over.

We can exploit the CPU-GPU biprocessor by interleaving code with a similar workload on both.
A kernel does not start until all previous kernels are over.

A **stream** is a new concept used for concurrent kernels.
Modifiers for the functions and launching executions on GPU:

Modifiers for the functions executed on GPU:

```c
__global__ void MyKernel() { }  // Invoked by the CPU
__device__ float MyFunc() { }  // Invoked by the GPU
```

Modifiers for the variables within GPU:

```c
__shared__ float MySharedArray[32];  // In shared mem.
__constant__ float MyConstantArray[32];
```

Configuration for the execution to launch kernels:

```c
dim2 gridDim(100,50);        // 5000 thread blocks
dim3 blockDim(4,8,8);        // 256 threads per blocks
MyKernel <<< gridDim,blockDim >>> (pars.);  // Launch
```

Note: We can see an optional third parameter here to indicate as a hint the amount of shared memory allocated dynamically by the kernel during its execution.
Programmer has to choose the block size and the number of blocks to exploit the maximum amount of parallelism for the code during its execution.
Functions to query at runtime the hardware resources we count on

Each GPU available at hardware level receives an integer tag which identifies it, starting in 0.

To know the number of GPUs available:

```c
cudaGetDeviceCount(int* count);
```

To know the resources available on GPU dev (cache, registers, clock frequency, ...):

```c
cudaGetDeviceProperties(struct cudaDeviceProp* prop, int dev);
```

To know the GPU that better meets certain requirements:

```c
cudaChooseDevice(int* dev, const struct cudaDeviceProp* prop);
```

To select a particular GPU:

```c
cudaSetDevice(int dev);
```

To know in which GPU we are executing the code:

```c
cudaGetDevice(int* dev);
```
The output of cudaGetDeviceProperties

This is exactly the output you get from the “DeviceQuery” code in the CUDA SDK.

There are 4 devices supporting CUDA

Device 0: "GeForce GTX 480"
  CUDA Driver Version: 4.0
  CUDA Runtime Version: 4.0
  CUDA Capability Major revision number: 2
  CUDA Capability Minor revision number: 0
  Total amount of global memory: 1609760768 bytes
  Number of multiprocessors: 15
  Number of cores: 480
  Total amount of constant memory: 65536 bytes
  Total amount of shared memory per block: 49152 bytes
  Total number of registers available per block: 32768
  Warp size: 32
  Maximum number of threads per block: 1024
  Maximum sizes of each dimension of a block: 1024 x 1024 x 64
  Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
  Maximum memory pitch: 2147483647 bytes
  Texture alignment: 512 bytes
  Clock rate: 1.40 GHz
  Concurrent copy and execution: Yes
  Run time limit on kernels: No
  Integrated: No
  Support host page-locked memory mapping: Yes
  Compute mode: Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution: Yes
  Device has ECC support enabled: No
Let’s manage video memory

To allocate and free GPU memory:
- `cudaMalloc(pointer, size)`
- `cudaFree(pointer)`

To move memory areas between CPU and GPU:
- On the CPU side, we declare `malloc(h_A)`.
- Also on the GPU side, we declare `cudaMalloc(d_A)`.
- And once this is done, we can:
  - Transfer data from the CPU to the GPU:
    - `cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);`
  - Transfer data from the GPU to the CPU:
    - `cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);`
- Prefix “h_” useful in practice as a tag for “host memory pointer”.
- Prefix “d_” also useful as a tag for “device (video) memory”.
IV. 2. A couple of examples
Example 1: What your code has to do

- Allocate N integers in CPU memory.
- Allocate N integers in GPU memory.
- Initialize GPU memory to zero.
- Copy values from GPU to CPU.
- Print values.
Example 1: Solution
[C code in red, CUDA extensions in blue]

```c
int main()
{
    int N = 16;
    int num_bytes = N*sizeof(int);
    int *d_a=0, *h_a=0; // Pointers in device (GPU) and host (CPU)

    h_a = (int*) malloc(num_bytes);
    cudaMemcpy( d_a, num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);
    for (int i=0; i<N; i++) printf("%d ", h_a[i]);
    free(h_a);
    cudaFree(d_a);
}
```
Asynchronous memory transfers

`cudaMemcpy()` calls are synchronous, that is:
- They do not start until all previous CUDA calls have finalized.
- The return to the CPU does not take place until we have performed the actual copy in memory.

From CUDA Compute Capabilities 1.2 on, it is possible to use the `cudaMemcpyAsync()` variant, which introduces the following differences:
- The return to the CPU is immediate.
- We can overlap computation and communication.
Example 2: Increment a scalar value “b” to the N elements of an array

The C program. This file is compiled with gcc

```c
void increment_cpu(float *a, float b, int N) {
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}

void main() {
    ..... 
    increment_cpu(a, b, N);
}
```

The CUDA kernel running on GPU followed by host code running on CPU. This file is compiled with nvcc

```c
__global__ void increment_gpu(float *a, float b, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main() {
    ..... 
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N/(float)blocksize));
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Example 2: Increment a scalar “b” to the N elements of a vector

Say N=16 and blockDim=4. Then we have 4 thread blocks, and each thread computes a single element of the vector. This is what we want: fine-grained parallelism for the GPU.

```
| blockIdx.x = 0 |
| blockIdx.x = 1 |
| blockIdx.x = 2 |
| blockIdx.x = 3 |
| blockDim.x = 4 |
| blockDim.x = 4 |
| blockDim.x = 4 |
| blockDim.x = 4 |
| threadIdx.x = 0,1,2,3 |
| threadIdx.x = 0,1,2,3 |
| threadIdx.x = 0,1,2,3 |
| threadIdx.x = 0,1,2,3 |
| idx = 0,1,2,3 |
| idx = 4,5,6,7 |
| idx = 8,9,10,11 |
| idx = 12,13,14,15 |
```

Language extensions:

```
int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
```

It will map from local index `threadIdx.x` to global index `idx`. 

Warning: blockDim.x should be \( \geq 32 \) (warp size), this is just an example.
More details for the CPU code of example 2
[red for C, green for variables, blue for CUDA]

// Reserve memory on the CPU
unsigned int numBytes = N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// Reserve memory on the GPU
float* d_A = 0; cudaMalloc((void**)&d_A, numbytes);

// Copy data from CPU to GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// Execute CUDA kernel with a number of blocks and block size
increment_gpu <<< N/blockSize, blockSize >>> (d_A, b);

// Copy data back to the CPU
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// Free video memory
cudaFree(d_A);
V. Compilation
void function_in_CPU(...) {
    ...
}
void other_funcs_CPU(int ...)
{
    ...
}
void saxpy_serial(float ...)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
void main() {
    float x;
    saxpy_serial(..);
    ...
}
A CUDA code is compiled with the NVCC compiler. NVCC separates CPU code and GPU code.

The compilation is a two step process:

- **Virtual**: Generates PTX (Parallel Thread eXecution).
- **Physical**: Generates the binary for a specific GPU (or even a CPU - more on this later).
The nvcc compiler and PTX virtual machine

C/C++ CUDA Application

EDG

Separates GPU and CPU code.

Open64

Generates PTX assembler.

Parallel Thread eXecution (PTX)

Virtual machine and ISA.

Programming model.

Resources and execution states.

CPU Code

PTX Code

float4 me = gx[gtid];
me.x += me.y * me.z;

ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;

float4 me = gx[gtid];
me.x += me.y * me.z;

ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
NVCC (NVidia CUDA Compiler)

NVCC is a compiler driver.
- Invokes all compilers and tools required, like cudacc, g++, cl, ...

NVCC produces two outputs:
- C code for the CPU, which must be compiled with the rest of the application using another compilation tool.
- PTX object code for the GPU.

Compilation process in Linux:

Compilation process in Windows:
Compile the kernel code with the -cubin flag to determine register usage.

On-line alternative: `nvcc --ptxas-options=-v`

Open the `.cubin` file with a text editor and look for the “code” section:

```plaintext
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
    name = myGPUcode
    lmem = 0
    smem = 68
    reg = 20
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09
        0x00200780
        ...
    }
}
```

- **Per thread:**
  - local memory
  - (used by compiler to spill registers to device memory)

- **Per thread-block:**
  - shared memory

- **Per thread:**
  - registers
Configuration for the execution: Heuristics

- The number of threads must be a multiple of warp size.
  - To avoid wasting computation on incomplete warps.

- The number of blocks must exceed the number of SMXs (1), and, if possible, double that number (2):
  1. So that each multiprocessor can have at least a block to work with.
  2. So that there is at least an active block which guarantees occupancy of that SMX when the block being executed suffers from a stall due to a memory access, unavailability of resources, bank conflicts, global stalls of all threads on a synchronization point (\texttt{__syncthreads()}), etc.

- Resources used by a block (register file and shared memory) must be at least half of the total available.
  - Otherwise, it is better to merge blocks.
General rules for the code to be scalable in future generations and for the blocks stream to be processed within a pipeline:

1. Think big for the number of blocks.
2. Think small for the size of threads.

Tradeoff: More threads per block means better memory latency hiding, but also means fewer registers per thread.

Hint: Use at least 64 threads per block, or even better, 128 or 256 threads (often there is still enough number of registers).

Tradeoff: Increasing occupancy does not necessarily mean higher performance, but the low occupancy for a SMX prevents from hide latency on memory bound kernels.

Hint: Pay attention to arithmetic intensity and parallelism.
Parametrization of an application

Everything related to performance is application-dependent, so you have to experiment for achieving optimal results.

GPUs may also vary in many ways depending on a particular model:

- Number of multiprocessors (SMs) and cores per SM.
- Memory bandwidth: From 100 GB/s to 500 GB/s.
- Register file size per SM: 8K, 16K, 32K (Fermi), 64K (Kepler).
- Shared memory size: 16 KB. per SM before Fermi, up to 48 KB. now.
- Threads: Check the per-block and the global limits.
  - Per-block: 512 (G80 and GT200), 1024 (Fermi and Kepler).
  - Total: 768 (G80), 1024 (GT200), 1536 (Fermi), 2048 (Kepler).
CUDA Occupancy Calculator

To help you select parameters for your application wisely

To reach the maximum degree of parallelism, use wisely the orange table of the tool (1)

The first row is the number of threads per block:

- The limit is 1024 in Fermi and Kepler generations.
- Power of two values are usually the best choices.
- List of potential candidates: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.
- We'll use 256 as first estimate, development cycles will tune the optimal value here, but usually:
  - Small values [2, 4, 8, 16] do not fully exploit the warp size and shared memory banks.
  - Intermediate values [32, 64] compromise thread cooperation and scalability in Kepler, Maxwell and future GPUs.
  - Large values [512, 1024] prevent from having enough number of concurrent blocks on each multiprocessor (the limits for the threads per block and per SMX are very close to each other). Also, the amount of registers per thread is too small.
To reach the maximum degree of parallelism, use wisely the orange table of the tool (2)

The second row is the number of registers per thread.

- We access the .cubin file to know this.
- The limit for each SM is 8K (G80), 16K (GT200), 32K (Fermi), 64K (Kepler), so when consuming 10 regs./thread is possible to execute:
  - On G80: 768 threads/SM, that is, 3 blocks of 256 thr \[3 \times 256 \times 10 = 7680\] (< 8192).
  - On Kepler: We reach the maximum of 2048 threads per SMX, but the use of registers is very low (we could have used up to 29 registers per thread):
    8 blocks \times 256 threads/block \times 10 registers/thread = 22480 regs. (< 65536 max.).
- In the G80 case, using 11 registers/thread, it would have meant to stay in 2 blocks, sacrificing 1/3 of parallelism => It is worth cutting that register down working more on the CUDA code for the thread.
To reach the maximum degree of parallelism, use wisely the orange table of the tool (3)

The third row is the shared memory spent for each block:

- We will also get this from the .cubin file, though we can carry out a manual accounting, as everything depends on where we put the `__shared__` prefix during memory declarations in our program.
- Limit: 16 KB (CCC 1.x), 16/48 KB (CCC 2.x), 16/32/48 KB (3.x).
- In the previous case for the G80, we won’t spend more than 5 KB of shared memory per block, so that we can reach the maximum of 3 concurrent blocks on each multiprocessor:
  - 3 blocks x 5 KB./block = 15 KB (< 16 KB.)
- With more than 5.34 KB. of shared memory used for each block, we sacrifice 33% of parallelism, the same performance hit than previously if we were unable of cutting down to 10 registers/thread.
VI. Examples: VectorAdd, Stencil, ReverseArray, MxM
Step for building the CUDA source code

1. Identify those parts with a good potential to run in parallel exploiting SIMD data parallelism.
2. Identify all data necessary for the computations.
3. Move data to the GPU.
4. Call to the computational kernel.
5. Establish the required CPU-GPU synchronization.
6. Transfer results from GPU back to CPU.
7. Integrate the GPU results into CPU variables.
Coordinated efforts in parallel are required

Parallelism is given by blocks and threads.

Threads within each block may require an explicit synchronization, as only within a warp it is guaranteed its joint evolution (SIMD). Example:

```c
a[i] = b[i] + 7;
syncthreads();
x[i] = a[i-1]; // The warp 1 read here the value of a[31], // which should have been written by warp 0 BEFORE
```

Kernel borders place implicit barriers:

- `Kernel1 <<<nbounds, nthreads>>> (a, b, c);`
- `Kernel2 <<<nbounds, nthreads>>> (a, b);`

Blocks can coordinate using atomic operations:

- Example: Increment a counter `atomicInc();`
VI. 1. Adding two vectors
The required code for the GPU kernel and its invocation from the CPU side

```c
// Add two vectors of size N: C[1..N] = A[1..N] + B[1..N]
// Each thread calculates a single component of the output vector
__global__ void vecAdd(float* A, float* B, float* C) {
    int tid = threadIdx.x + (blockDim.x* blockIdx.x);
}
```

```c
int main() { // Launch N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

The `__global__` prefix indicates that `vecAdd()` will execute on device (GPU) and will be called from host (CPU).

A, B and C are pointers to device memory, so we need to:
- Allocate/free memory on GPU, using `cudaMalloc()`/`cudaFree()`.
- These pointers cannot be dereferenced in host code.
unsigned int numBytes = N * sizeof(float);
// Allocates CPU memory
float* h_A = (float*) malloc(numBytes);
float* h_B = (float*) malloc(numBytes);
... initializes h_A and h_B ...
// Allocates GPU memory
float* d_A = 0;  cudaMalloc((void**)&d_A, numBytes);
float* d_B = 0;  cudaMalloc((void**)&d_B, numBytes);
float* d_C = 0;  cudaMalloc((void**)&d_C, numBytes);
// Copy input data from CPU into GPU
cudaMemcopy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
cudaMemcopy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
... CALL TO THE VecAdd KERNEL IN THE PREVIOUS SLIDE HERE...
// Copy results from GPU back to CPU
float* h_C = (float*) malloc(numBytes);
cudaMemcopy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
// Free video memory
cudaFree(d_A);  cudaFree(d_B);  cudaFree(d_C);
Running in parallel (regardless of hardware generation)

- `vecAdd <<< 1, 1 >>>` ( ) Executes 1 block composed of 1 thread - no parallelism.

- `vecAdd <<< B, 1 >>>` ( ) Executes B blocks composed on 1 thread. Inter-multiprocessor parallelism.

- `vecAdd <<< B, M >>>` ( ) Executes B blocks composed of M threads each. Inter- and intra-multiprocessor parallelism.
With M threads per block, a unique index is given by:

$$tid = threadIdx.x + blockDim.x \times blockIdx.x;$$

Consider indexing an array of one element per thread (because we are interested in fine-grained parallelism), B=4 blocks of M=8 threads each:

Which thread will compute the 22nd element of the array?

gridDim.x is 4. blockDim.x is 8. blockIdx.x = 2. threadIdx.x = 5.

tid = 5 + (8 \times 2) = 21 (we start from 0, so this is the 22nd element).
Handling arbitrary vector sizes

Typical problems are not friendly multiples of blockDim.x, so we have to prevent accessing beyond the end of arrays:

```c
// Add two vectors of size N: C[1..N] = A[1..N] + B[1..N]
__global__ void vecAdd(float* A, float* B, float* C, int N) {
    int tid = threadIdx.x + (blockDim.x * blockIdx.x);
    if (tid < N)
}
```

And now, update the kernel launch to include the "incomplete" block of threads:

```c
vecAdd<<< (N + M-1)/256, 256>>>(d_A, d_B, d_C, N);
```
VI. 2. Stencil kernels
Rationale

Looking at the previous example, threads add a level of complexity without contributing with new features. However, unlike parallel blocks, threads can:
- Communicate (via shared memory).
- Synchronize (for example, to preserve data dependencies).

We need a more sophisticated example to illustrate all this...
1D Stencil

Consider applying a 1D stencil to a 1D array of elements. Each output element is the sum of input elements within a radius. If radius is 3, then each output element is the sum of 7 input elements:

Again, we apply fine-grained parallelism for each thread to process a single output element.

Input elements are read several times: With radius 3, each input element is read seven times.
Sharing data between threads. Advantages

Threads within a block can share data via shared memory.

- Shared memory is user-managed: Declare with __shared__ prefix.
- Data is allocated per block.
- Shared memory is extremely fast:
  - 500 times faster than global memory (video memory - GDDR5). The difference is technology: static (built with transistors) versus dynamic (capacitors).
  - Programmer can see it like an extension of the register file.
- Shared memory is more versatile than registers:
  - Registers are private to each thread, shared memory is private to each block.
Sharing data between threads. Limitations

- Shared memory and registers usage limit parallelism.
  - If we leave room for a second block, register file and shared memory are partitioned (even though blocks do not execute simultaneously, context switch is immediate).

- Examples for Kepler were shown before (for a max. of 64K registers and 48 Kbytes of shared memory per multiproc.):
  - To allocate two blocks per multiprocessor: The block cannot use more than 32 Kregisters and 24 Kbytes of shared memory.
  - To allocate three blocks per multiprocessor: The block cannot use more than 21.3 Kregisters and 16 Kbytes of shared memory.
  - To allocate four blocks per multiprocessor: The block cannot use more than 16 Kregisters and 12 Kbytes of shared memory.
  - ... and so on. Use the CUDA Occupancy Calculator to figure it out.
Steps to cache data in shared memory:

- Read \((\text{blockDim}.x + 2 \times \text{radius})\) input elements from global memory to shared memory.
- Compute \(\text{blockDim}.x\) output elements.
- Write \(\text{blockDim}.x\) output elements to global memory.

Each block needs a halo of \(\text{radius}\) elements at each boundary.
__global__ void stencil_1d(int *d_in, int *d_out)
{
   __shared__ int temp[ BLOCKSIZE + 2 * RADIUS ];
   int gindex = threadIdx.x
               + blockIdx.x * blockDim.x;
   int lindex = threadIdx.x + RADIUS;

   // Read input elements into shared memory
   temp[lindex] = d_in[gindex];
   if (threadIdx.x < RADIUS) {
      temp[lindex-RADIUS] = d_in[gindex-RADIUS];
      temp[lindex+blockDim.x]=d_in[gindex+blockDim.x];
   }

   // Apply the stencil
   int result = 0;
   for (int offset=-RADIUS; offset<=RADIUS; offset++)
      result += temp[lindex + offset];

   // Store the result
   d_out[gindex] = result;
}

But we have to prevent race conditions. For example, last thread reads the halo before first thread (from a different warp) has fetched it. Synchronization among threads is required!
Threads synchronization

Use __syncthreads() to synchronize all threads within a block:

- All threads must reach the barrier before progressing.
- This can be used to prevent RAW / WAR / WAW hazards.
- In conditional code, the condition must be uniform across the block.

```c
__global__ void stencil_1d(...) {
    < Declare variables and indices >
    < Read input elements into shared memory >

    __syncthreads();

    < Apply the stencil >
    < Store the result >
}
```
Summary of major concepts applied during this example

- Launch N blocks with M threads per block to execute threads in parallel. Use:
  ```
  kernel <<< N, M >>> ();
  ```

- Access block index within grid and thread index within block:
  ```
  blockIdx.x and threadIdx.x;
  ```

- Calculate global indices where each thread has to work depending on data partitioning. Use:
  ```
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```

- Declare a variable/array in shared memory. Use:
  ```
  __shared__ (as prefix to the data type).
  ```

- Synchronize threads to prevent data hazards. Use:
  ```
  __syncthreads();
  ```
VI. 3. Reverse the order of a vector of elements
GPU code for the ReverseArray kernel (1) using a single block

```c
__global__ void reverseArray(int *in, int *out) {
    int index_in = threadIdx.x;
    int index_out = blockDim.x - 1 - threadIdx.x;

    // Reverse array contents using a single block
    out[index_out] = in[index_in];
}
```

It is a solution too naive, which does not aspire to apply massive parallelism. The maximum block size is 1024 threads, so that is the largest vector that this code would accept as input.
GPU code for the ReverseArray kernel (2) using multiple blocks

```c
__global__ void reverseArray(int *in, int *out) {
    int in_offset = blockIdx.x * blockDim.x;
    int out_offset = (gridDim.x - 1 - blockIdx.x) * blockDim.x;
    int index_in = in_offset + threadIdx.x;
    int index_out = out_offset + (blockDim.x - 1 - threadIdx.x);

    // Reverse contents in chunks of whole blocks
    out[index_out] = in[index_in];
}
```
A more sophisticated version using shared memory

Input addresses are linear and aligned = coalesced

Input Global Mem

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

- blockOffset=0
- blockOffset=4
- blockOffset=8
- blockOffset=12

Registers
(local variables)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Shared Memory

| 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 |

Output Global Mem

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Output addresses are linear and aligned = coalesced
GPU code for the ReverseArray kernel (3) using multiple blocks and shared memory

```c
__global__ void reverseArray(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    syncthreads();

    // Reverse local arrays within each block
    temp[lindex] = temp[blockDim.x-lindex-1];
    syncthreads();

    // Reverse contents in chunks of whole blocks
    out[threadIdx.x + ((N/blockDim.x)-blockIdx.x-1) * blockDim.x] = temp[lindex];
}
```
VI. 4. Matrix product
Typical CPU code written in C language

\[ C = A \times B. \]

All square matrices of size \( N \times N \).

Matrices are serialized into vectors to simplify dynamic memory allocation.

```c
void MxMonCPU(float* A, float* B, float* C, int N);
{
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            {
                float sum=0;
                for (int k=0; k<N; k++)
                    {
                        float a = A[i*N + k];
                        float b = B[k*N + j];
                        sum += a*b;
                    }
                C[i*N + j] = sum;
            }
}
```
CUDA version for the matrix product:  
A draft for the parallel code

```c
__global__ MxMonGPU(float* A, float* B, float* C, int N) {
    float sum=0;
    int i, j;
    i = threadIdx.x + blockIdx.x * blockDim.x;
    j = threadIdx.y + blockIdx.y * blockDim.y;
    for (int k=0; k<N; k++)
        { float a = A[i*N + k];
          float b = B[k*N + j];
          sum += a*b;
        }
    C[i*N + j] = sum;
}
```
Each thread computes a single element of C.

Matrices A and B are loaded N times from video memory.

Blocks accommodate threads in groups of 1024 threads (internal CUDA constraint in Fermi and Kepler). That way, we may use 2D blocks composed of 32x32 threads each.

```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(WidthB/BLOCKSIZE, HeightA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid, dimBlock>>> (A, B, C, N);
```
CUDA version for the matrix product: 

### Analysis

- Each thread requires 10 registers, so we can reach the maximum amount of parallelism in Kepler:
  - 2 blocks of 1024 threads (32x32) on each SMX. (2x1024x10 = 20480 registers, which is lower than 65536 registers available).

- **Problems:**
  - Low arithmetic intensity.
  - Demanding on memory bandwidth, which becomes the bottleneck.

- **Solution:**
  - Use shared memory on each multiprocessor.
Using shared memory: Version with tiling for A and B

The 32x32 submatrix $C_{sub}$ computed by each thread block uses tiles of 32x32 elements of A and B which are repeatedly allocated on shared memory.

A and B are loaded only $(N/32)$ times from global memory.

Achievements:
- Less demanding on memory bandwidth.
- More arithmetic intensity.
We have to manage all tiles involved within a thread block:

Load **in parallel** (all threads contribute) the input tiles (A and B) from global memory into shared memory. Tiles reuse the shared memory space.

`__syncthreads()` (to make sure we have loaded matrices before starting the computation).

Compute all products and sums for C using tiles within shared memory.

Each thread can now iterate independently on tile elements.

`__syncthreads()` (to make sure that the computation with the tile is over before loading, in the same memory space within share memory, two new tiles of A and B in the next iteration).
A trick to avoid shared memory bank conflicts

Rationale:

- The shared memory is structured into 16 (pre-Fermi) or 32 banks.
- Threads within a block are numbered in column major order, that is, the x dimension is the fastest varying.

When using the regular indexing scheme to shared memory arrays: `shData[threadIdx.x][threadIdx.y]`, threads within a half-warp will be reading from the same column, that is, from the same shared memory bank.

However, using `shData[threadIdx.y][threadIdx.x]`, threads within a half-warp will be reading from the same row, which implies reading from a different bank each.

So, tiles store/access data in shared memory transposed.
Tiling: The CUDA code for the GPU kernel

__global__ void MxMonGPU(float *A, float *B, float *C, int N)
{
    int sum=0, tx, ty, i, j;
    tx = threadIdx.x; ty = threadIdx.y;
    i = tx + blockIdx.x*blockDim.x; j = ty + blockIdx.y*blockDim.y;
    __shared__ float As[32][32], float Bs[32][32];

    // Traverse tiles of A and B required to compute the block submatrix for C
    for (int tile=0; tile<(N/32); tile++)
    {
        // Load tiles (32x32) from A and B in parallel (and store them transposed)
        As[ty][tx] = A[(i*N) + (ty+(tile*32))];
        Bs[ty][tx] = B[((tx+(tile*32))*N) + j];
        __syncthreads();

        // Compute results for the submatrix of C
        for (int k=0; k<32; k++) // Data have to be read from tiles transposed too
            sum += As[k][tx] * Bs[ty][k];
        __syncthreads();
    }

    // Write all results for the block in parallel
    C[i*N+j] = sum;
}
A compiler optimization: Loop unrolling

Without loop unrolling:

```c
... 
__syncthreads();

// Compute results for that tile
for (k=0; k<32; k++)
    sum += As[tx][k]*Bs[k][ty];
__syncthreads();
}
C[indexC] = sum;
```

Unrolling the loop:

```c
... 
__syncthreads();

// Compute results for that tile
sum += As[tx][0]*Bs[0][ty];
sum += As[tx][1]*Bs[1][ty];
sum += As[tx][2]*Bs[2][ty];
sum += As[tx][3]*Bs[3][ty];
sum += As[tx][4]*Bs[4][ty];
sum += As[tx][5]*Bs[5][ty];
sum += As[tx][6]*Bs[6][ty];
sum += As[tx][7]*Bs[7][ty];
sum += As[tx][8]*Bs[8][ty];
    ...
sum += As[tx][31]*Bs[31][ty];
__syncthreads();
}
C[indexC] = sum;
```
Performance on the G80 for tiling & unrolling

Tile size (32x32 unfeasible on G80 hardware)
VII. Bibliography and tools
CUDA Zone: Basic web resource for a CUDA programmer

[developer.nvidia.com/cuda-zone]
CUDA 6 Production Release. Free download for all platforms and users

[developer.nvidia.com/cuda-downloads]

### Windows, Linux, Mac OSX

<table>
<thead>
<tr>
<th>Version</th>
<th>64-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 7</td>
<td>Notebook</td>
<td>EXE</td>
</tr>
<tr>
<td>Windows Vista</td>
<td>Desktop</td>
<td>EXE</td>
</tr>
<tr>
<td>Windows XP</td>
<td>Desktop</td>
<td>EXE</td>
</tr>
</tbody>
</table>

Getting Started Guide

### Mac OSX

<table>
<thead>
<tr>
<th>OSX Release</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.8</td>
<td>PKG</td>
</tr>
<tr>
<td>10.9</td>
<td></td>
</tr>
</tbody>
</table>

Getting Started Guide

### Distribution

<table>
<thead>
<tr>
<th>Distribution</th>
<th>x86 64-bit</th>
<th>x86 32-bit</th>
<th>ARMv7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fedora 19</td>
<td>RPM</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>OpenSUSE 12.3</td>
<td>RPM</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>RHEL 6</td>
<td>RPM</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>RHEL 5</td>
<td></td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>CentOS 6</td>
<td>RPM</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>CentOS 5</td>
<td></td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>SLES 11 (SP2 &amp; SP3)</td>
<td>RPM</td>
<td>RUN</td>
<td></td>
</tr>
<tr>
<td>Ubuntu 13.04</td>
<td>DEB</td>
<td>RUN</td>
<td>RUN</td>
</tr>
<tr>
<td>Ubuntu 12.04</td>
<td>DEB</td>
<td>RUN</td>
<td>DEB</td>
</tr>
<tr>
<td>L4T, Linux for Tegra</td>
<td></td>
<td>LINK</td>
<td>LINK</td>
</tr>
</tbody>
</table>
CUDA books: From 2007 to 2013

GPU Gems series [developer.vidia.com/content/GPUGems3/gpugems3_part01.html]

List of CUDA books in [www.nvidia.com/object/cuda_books.html]

Sep'07 | Feb'10 | Jul'10 | Abr'11 | Oct'11
---|---|---|---|---
Nov'11 | Dic'12 | Jun'13 | Oct'13 | Sep'14
Guides for developers and more documents

- Getting started with CUDA C: Programmers guide.
  [docs.nvidia.com/cuda/cuda-c-programming-guide]

- For tough programmers: The best practices guide.
  [docs.nvidia.com/cuda/cuda-c-best-practices-guide]

- The root web collecting all CUDA-related documents:
  [docs.nvidia.com/cuda]

where we can find, additional guides for:
- Installing CUDA on Linux, MacOS and Windows.
- Optimize and improve CUDA programs on Kepler platforms.
- Check the CUDA API syntax (runtime, driver and math).
- Learn to use libraries like cuBLAS, cuFFT, cuRAND, cuSPARSE, ...
- Deal with basic tools (compiler, debugger, profiler).
Choices to accelerate your applications on GPUs and material for teaching CUDA

[developer.nvidia.com/cuda-education-training] (also available from the left lower corner of the CUDA Zone)

CUDA Education & Training

Accelerate Your Applications
Learn using step-by-step instructions, video tutorials and code samples.
- Accelerated Computing with C/C++
- Accelerated Applications on GPUs with OpenACC Directives
- Accelerated Numerical Analysis Tools with GPUs
- Drop-in Acceleration on GPUs with Libraries
- GPU Accelerated Computing with Python

Teaching Resources
Get the latest educational slides, hands-on exercises and access to GPUs for your parallel programming courses.
- Parallel Programming Training Materials
- NVIDIA Research & Academic Programs

Sign up to join the Accelerated Computing Educators Network. This network seeks to provide a collaborative area for those looking to educate others on massively parallel programming. Receive updates on new educational material, access to CUDA Cloud Training Platforms, special events for educators, and an educators focused news letter.

Sign-up Today!

Tweets by @GPUComputing

QUICKLINKS
- Downloads
- CUDA GPUs
- NVIDIA Nsight Visual Studio Edition
- Get Started - Parallel Computing
- Tools & Ecosystem
- CUDA FAQ

Manuel Ujaldon - Nvidia CUDA Fellow
Courses on-line (free access)

More than 50,000 registered users from 127 countries over the last 6 months. An opportunity to learn from CUDA masters:

- Prof. Wen-Mei Hwu (Univ. of Illinois).
- Prof. John Owens (Univ. of California at Davis).
- Dr. David Luebke (Nvidia Research).

There are two basic options, both recommended:

- **Introduction to parallel programming:**
  - 7 units of 3 hours = 21 hours.
  - Provides high-end GPUs to carry out the proposed assignments.
  - [https://developer.nvidia.com/udacity-cs344-intro-parallel-programming]

- **Heterogeneous Parallel Programming:**
  - 9 weeks, each with classes (20’ video), quizzes and programming assignments.
  - [https://www.coursera.org/course/hetero]
Tutorials about C/C++, Fortran and Python

You have to register on the Amazon EC2 services available on the Web (cloud computing): [nvidia.qwiklab.com]

They are usually sessions of 90 minutes.

Only a Web browser and SSH client are required.

Some tutorials are free, other require tokens of $29.99.
Talks and webinars

Talks recorded at GTC (Graphics Technology Conference):
- 383 talks from 2013.
- More than 500 available from 2014.
- [www.gputechconf.com/gtcnew/on-demand-gtc.php]

Webinars about GPU computing:
- List of past talks on video (mp4/wmv) and slides (PDF).
- List of incoming on-line talks to be enrolled.
- [developer.nvidia.com/gpu-computing-webinars]

CUDACasts:
- [bit.ly/cudacasts]
Examples of webinars about CUDA 6.0

GTC Express Webinar Program

GTC Express is a year-round extension of the great content available at GTC. Each month, top developers, scientists, researchers, and industry practitioners present innovative and thought-provoking webinars focused on the GPU-enabled work they’re doing and sharing how GPUs are transforming their fields.

Register below for upcoming webinars and explore recordings of previous webinars.

**GTC EXPRESS SCHEDULE AT-A-GLANCE**

<table>
<thead>
<tr>
<th>Date</th>
<th>Title</th>
<th>Speaker</th>
<th>Register Now</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 3, 2014, 9:00 AM PDT</td>
<td>The Next Steps for Folding@home</td>
<td>Vijay Pande, Professor, Stanford University</td>
<td><a href="#">Register Now</a></td>
</tr>
<tr>
<td>May 14, 2014, 10:00 AM PDT</td>
<td>CUDA 6: Performance Overview</td>
<td>Jonathan Cohen, Senior Manager, CUDA Libraries and Algorithms, NVIDIA</td>
<td><a href="#">Register Now</a></td>
</tr>
<tr>
<td>May 13, 2014, 9:00 AM PDT</td>
<td>An Overview of AMBER 14 - Creating the World's Fastest Molecular Dynamics Software Package</td>
<td>Ross C. Walker, University of California San Diego, Scott Le Grand, Amazon Web Services, and Adrian Roltberg, University of Florida</td>
<td><a href="#">Register Now</a></td>
</tr>
<tr>
<td>May 7, 2014, 10:00 AM PDT</td>
<td>CUDA 6: Drop-in Performance Optimized Libraries</td>
<td>NVIDIA DevTech Team</td>
<td><a href="#">Register Now</a></td>
</tr>
<tr>
<td>May 1, 2014, 10:00 AM PDT</td>
<td>CUDA 6: Unified Memory</td>
<td>Mark Ebersole, CUDA Educator, NVIDIA</td>
<td><a href="#">Register Now</a></td>
</tr>
<tr>
<td>April 23, 2014, 11:00 AM PDT</td>
<td>CUDA 6 Features Overview</td>
<td>Ujval Kapasi, CUDA Product Manager, NVIDIA</td>
<td><a href="#">Register Now</a></td>
</tr>
</tbody>
</table>
Developers

Sign up as a registered developer:
- [www.nvidia.com/paralleldeveloper]
- Access to exclusive developer downloads.
- Exclusive access to pre-release CUDA installers like CUDA 6.0.
- Exclusive activities and special offers.

Meeting point with many other developers:
- [www.gpucomputing.net]

GPU news and events:
- [www.gpgpu.org]

Technical questions on-line:
- NVIDIA Developer Forums: [devtalk.nvidia.com]
- Search or ask on: [stackoverflow.com/tags/cuda]
Developers (2)

List of CUDA-enabled GPUs:
[developer.nvidia.com/cuda-gpus]

And a last tool for tuning code: The CUDA Occupancy Calculator
[developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls]
Future developments

Nvidia’s blog contains articles unveiling future technology to be used within CUDA. It is the most reliable source about what’s next (subscription recommended):

[devblogs.nvidia.com/parallelforkall]

Some recommended articles:

“5 Powerful New Features in CUDA 6”, by Mark Harris.
“CUDA 6.0 Unified Memory”, by Mark Ebersole.
“CUDA Dynamic Parallelism API and Principles”, by Andrew Adinetz.
“NVLINK, Pascal and Stacked Memory: Feeding the Appetite for Big Data”, by Denis Foley.
“CUDA Pro Tip: Increase Application Performance with NVIDIA GPU Boost”, by Mark Harris.
Thanks for your attention!

You can always reach me in Spain at the Computer Architecture Department of the University of Malaga:

- e-mail: ujaldon@uma.es
- Phone: +34 952 13 28 24.

Or, more specifically on GPUs, visit my web page as Nvidia CUDA Fellow:

http://research.nvidia.com/users/manuel-ujaldon