Programming Graphic Processing Units with CUDA

Paolo Burgio
paolo.burgio@unimore.it
✓ (Co-)processor devoted to graphics
  – Built as "monolithical" chip
  – Integrated as co-processor
  – Recently, SoCs

✓ Main providers
  – NVIDIA
  – ATI
  – AMD
  – Intel…

✓ We will focus on NVIDIA
  – Widely adopted
  – Adopted by us
A bit of history…

✓ 70s: first "known" graphic card on a board package
✓ Early 90s: 3D graphics popular in games
✓ 1992: OpenGL
✓ 1999: NVIDIA GeForce 256 "World's first GPU"
✓ 2001: NVIDIA GeForce 3, w/programmable shaders (First GP-GPU)
✓ 2008: NVIDIA GeForce 8800 GTX w/CUDA capabilities - Tesla arch.
✓ 2009: OpenCL 1.0 inside MAC OS X Snow Leopard
✓ 2010: NVIDIA GeForce 400 Series - Fermi arch.
✓ 2010-1: OpenCL 1.1, 1.2
✓ 2012: NVIDIA GeForce 600 Series - Kepler arch.
✓ 2013: OpenCL 2.0
✓ 2015 Q4: NVIDIA and HiPeRT Lab start cooperation ;)
✓ 2017 Q1: NVIDIA Drive Px2 for Self-Driving Cars
✓ Many architectures
  – Tesla, Fermi, Maxwell, (soon) Parker..

✓ Many programming libraries languages frameworks
  – OpenGL
  – CUDA
  – OpenCL
  – …

✓ Many application domains!
  – Graphics
  – GP-GPUs?
  – Automotive!??!?!??!

✓ Let's start from scratch…
GPU for graphics - OpenGL

✓ Use GPUs for rendering of graphics
  – A library of functions and datatypes
  – Use directly in the code
  – High-level operations on lights, shapes, shaders…

✓ Tailored for the specific domain and programmer skills
  – Hides away the complexity of the machine
  – Takes care of "low" level optimizations/operations
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("blender");
    glutDisplayFunc(display);
    glutVisibilityFunc(visible);

    glNewList(1, GL_COMPILE); /* create ico display list */
    glutSolidIcosahedron();
    glEndList();

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT2, GL_POSITION, light2_position);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_LINE_SMOOTH);
    glLineWidth(2.0);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(/* field of view in degree */ 40.0,
        /* aspect ratio */ 1.0,
        /* Z near */ 1.0,
        /* Z far */ 10.0);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 5.0, /* eye is at (0,0,5) */
        0.0, 0.0, 0.0, /* center is at (0,0,0) */
        0.0, 1.0, 0.0); /* up is in positive Y direction */
    glTranslatef(0.0, 0.6, -1.0);

    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("blender");
    glutDisplayFunc(display);
    glutVisibilityFunc(visible);

    glNewList(1, GL_COMPILE); /* create ico display list */
    glutSolidIcosahedron();
    glEndList();

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT2, GL_POSITION, light2_position);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_LINE_SMOOTH);
    glLineWidth(2.0);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(/* field of view in degree */ 40.0,
                    /* aspect ratio */ 1.0,
                    /* Z near */ 1.0,
                    /* Z far */ 10.0);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 5.0, /* eye is at (0,0,5) */
              0.0, 0.0, 0.0, /* center is at (0,0,0) */
              0.0, 1.0, 0.0); /* up is in positive Y direction */
    glTranslatef(0.0, 0.6, -1.0);

    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("blender");
    glutDisplayFunc(display);
    glutVisibilityFunc(visible);

    glNewList(1, GL_COMPILE); /* create ico display list */
    glutSolidIcosahedron();
    glEndList();

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT2, GL_POSITION, light2_position);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_LINE_SMOOTH);

    glLineWidth(2.0);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(/* field of view in degree */ 40.0,
                   /* aspect ratio */ 1.0,
                   /* Z near */ 1.0,
                   /* Z far */ 10.0);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 5.0, /* eye is at (0,0,5) */
              0.0, 0.0, 0.0, /* center is at (0,0,0) */
              0.0, 1.0, 0.0); /* up is in positive */
    glTranslatef(0.0, 0.6, -1.0);

    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}
```c
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("icosahedron");
    glutDisplayFunc(display);
    glutVisibilityFunc(visible);

    glNewList(1, GL_COMPILE); /* create ico display list */
    glSolidIcosahedron();
    glEndList();

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_AMBIENT, 1.0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, 1.0);
    glLightfv(GL_LIGHT0, GL_POSITION, 3.0);
    glLightfv(GL_LIGHT2, GL_DIFFUSE, 1.0);
    glLightfv(GL_LIGHT2, GL_POSITION, 3.0);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_LINE_SMOOTH);
    glLineWidth(2.0);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(/* field of view in radians */ /* aspect ratio */ /* Z near */ /* Z far */);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 5.0, /* eye is */ 0.0, 0.0, 0.0, /* center is */ 0.0, 1.0, 0.0, /* up is */
               0.0, 0.6, -1.0);

    glutMainLoop();
    return 0; /* ANSI C requires main */
}

void display(void) {
    static GLfloat amb[] = {0.4, 0.4, 0.4, 0.0};
    static GLfloat dif[] = {1.0, 1.0, 1.0, 0.0};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_LIGHT1);
    glDisable(GL_LIGHT2);
    amb[3] = dif[3] = cos(s) / 2.0 + 0.5;
    glMaterialfv(GL_FRONT, GL_AMBIENT, amb);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, dif);

    glPushMatrix();
    glTranslatef(-0.3, -0.3, 0.0);
    glRotatef(angle1, 1.0, 5.0, 0.0);
    glCallList(1); /* render ico display list */
    glPopMatrix();

    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_LIGHT2);
    glDisable(GL_LIGHT1);
    amb[3] = dif[3] = 0.5 - cos(s * 0.95) / 2.0;
    glMaterialfv(GL_FRONT, GL_AMBIENT, amb);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, dif);

    glPushMatrix();
    glTranslatef(0.3, 0.3, 0.0);
    glRotatef(angle2, 1.0, 0.0, 5.0);
    glCallList(1); /* render ico display list */
    glPopMatrix();
}
```

GPU for graphics - OpenGL

✓ Use GPUs for rendering of graphics
  – A library of functions and datatypes
  – Use directly in the code
  – High-level operations on lights, shapes, shaders…

✓ Tailored for the specific domain and programmer skills
  – Hides away the complexity of the machine
  – Takes care of "low" level optimizations/operations
General Purpose - GPUs

✔️ We have a machine with thousand of cores
  - why should we use it only for graphics?

✔️ Use it for General Purpose Computing!
  - GP-GPU
  - ~yr 2000

NdA: Computing modes
  - General Purpose Computing
  - ...

©2017 University of Modena and Reggio Emilia
General Purpose - GPUs

✓ We have a machine with thousands of cores – why should we use it only for graphics?

✓ Use it for General Purpose Computing!
  – GP-GPU
  – ~yr 2000

NdA: Computing modes
  – General Purpose Computing
  – High-Performance Computing
  – ...

©2017 University of Modena and Reggio Emilia Parallel Programming LM – 2017/18
General Purpose - GPUs

✓ We have a machine with thousand of cores
  – why should we use it only for graphics?

✓ Use it for General Purpose Computing!
  – GP-GPU
  – ~yr 2000

NdA: Computing modes
  – General Purpose Computing
  – High-Performance Computing
  – Embedded Computing
  – …
✓ We have a machine with thousand of cores
  – why should we use it only for graphics?

✓ Use it for General Purpose!
  – GP-GPU
  – ~yr 2000

NdA: Computing modes
  – General Purpose Computing
  – High-Performance Computing
  – Embedded Computing
  – Real-Time Computing
  – …
Under the hood: face detection
Under the hood: face detection

Face detection

©2017 University of Modena and Reggio Emilia
Parallel Programming LM – 2017/18
Under the hood: face detection
Under the hood: face detection

- Color img -> Gray img
- Histogram equalization
- Smoothing
- Sobel filter + Detection
- Canny filter
- Detection
- Faces
Under the hood: face detection

- Color img -> Gray img
- Histogram equalization
- Smoothing
- Sobel filter + Detection
- Canny filter
- Detection
- Faces
Under the hood: face detection

- Color img -> Gray img
- Histogram equalization
- Smoothing
- Sobel filter + Detection
- Canny filter
- Detection

Face detection
Under the hood: face detection

- Color img -> Gray img
- Histogram equalization
- Smoothing
- Sobel filter + Detection
- Canny filter
- Detection
- Faces
Under the hood: face detection

- Color img -> Gray img
- Histogram equalization
- Smoothing
- Sobel filter + Detection
- Canny filter
- Detection
- Faces
Under the hood: face detection

1. Color img -> Gray img
2. Histogram equalization
3. Smoothing
4. Sobel filter + Detection
5. Canny filter
6. Detection

**Face Features**
Image binarization

✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
  - unsigned chars
  - 255 => white
  - 0 => black
✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
  - unsigned chars
  - 255 => white
  - 0 => black

Image binarization
Image binarization

✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
  - unsigned chars
  - 255 => white
  - 0 => black

```c
#define GRAY_THRESHOLD 100
#define WHITE 255
#define BLACK 0

void binarizeImage(const unsigned char inputImg[],
                    unsigned char outputImg[],
                    unsigned int imgDim)
{
    for(int i=0; i<imgDim; i++)
        if(inputImg[i] >= GRAY_THRESHOLD)
            outputImg[i] = WHITE;
        else
            outputImg[i] = BLACK;
}
```
Image binarization

✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
   - unsigned char
   - 255 => white
   - 0 => black

```c
#define GRAY_THRESHOLD 100
#define WHITE 255
#define BLACK 0

void binarizeImage(const unsigned char inputImg[],
                    unsigned char outputImg[],
                    unsigned int imgDim)
{
    for(int i=0; i<imgDim; i++)
        if(inputImg[i] >= GRAY_THRESHOLD)
            outputImg[i] = WHITE;
        else
            outputImg[i] = BLACK;
}
```
Image binarization

✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
  - unsigned chars
  - 255 => white
  - 0 => black

```c
#define GRAY_THRESHOLD 100
#define WHITE 255
#define BLACK 0

void binarizeImage(const unsigned char inputImg[],
                    unsigned char outputImg[],
                    unsigned int imgDim)
{
    for(int i=0; i<imgDim; i++)
        if(inputImg[i] >= GRAY_THRESHOLD)
            outputImg[i] = WHITE;
        else
            outputImg[i] = BLACK;
}
```
Image binarization

✓ Graylevel image => B/W image
✓ Pixel: 256 shades of gray
  - unsigned char
  - 255 => white
  - 0 => black

```c
#define GRAY_THRESHOLD 100
#define WHITE 255
#define BLACK 0

void binarizeImage(const unsigned char inputImg[],
                    unsigned char outputImg[],
                    unsigned int imgDim)
{
    for(int i=0; i<imgDim; i++)
        if(inputImg[i] >= GRAY_THRESHOLD)
            outputImg[i] = WHITE;
        else
            outputImg[i] = BLACK;
}
```

Multiple Data
Image binarization

- Graylevel image => B/W image
- Pixel: 256 shades of gray
  - unsigned chars
  - 255 => white
  - 0 => black

```c
#define GRAY_THRESHOLD 100
#define WHITE 255
#define BLACK 0

void binarizeImage(const unsigned char inputImg[],
                    unsigned char outputImg[],
                    unsigned int imgDim) {
  for(int i=0; i<imgDim; i++)
    if(inputImg[i] >= GRAY_THRESHOLD)
      outputImg[i] = WHITE;
    else
      outputImg[i] = BLACK;
}
```
Let's (re)design them!

We want to perform graphics
  – E.g., filters, shaders…

Ultimately, operations on pixels!
  – Same algorithm repeated for each (subset of) pixels

Algorithm => program

(subset of) pixels => data

Same (single) Program, Multiple Data – SPMD
  – Not SIMD!
Algorithms for image processing are

- Highly regular (loop-based, with well known boundaries at image rows/columns)
- Massively parallel (thousands of threads)

Regular, "big" loops

- Single Program (Loop Iteration) Multiple Data - SPMD
- Parallel threads perform the very same operation on adjacent data

We need a massively parallel machine

- Thousands of cores

With simple cores

- FP Support

To perform the very same instruction!

- Same Fetch Unit and Decode Unit
Fetch and decode units

✓ Traditional pipeline

✓ Traditional parallel pipeline
GPU multi-core

✓ Share FU, DU, MEM units
  – Approximate scheme!

CUDA Cores

Shared Multiprocessor SM

FU → DU → LD

EX

EX

EX

MEM
SMs as building block

✓ Architecture of the SM
   – GPU "class"
   – Kepler has 192 cores
   – Maxwell/Pascal has 128 cores

✓ Number of SMs
   – GPU model
   – Maxwell's GTX980 has 10
   – Pascal's GTX1080 has 20
   – Pascal's Drive PX1 has 2

✓ NUMA memory system
GPU as a device

- Host-device scheme
- Hierarchical NUMA space
  - Non-Uniform Mem Access
Something you are less used to

GP-GPU based embedded platforms
✓ …this is not under your nose….  
✓ Still, host + accelerator model
✓ Communicate via shared memory
  – No PCI-express
  – Host memory "pull"
  – FPGA mem/BRAM "push"
To summarize…

✓ Tightly-coupled SMs
  – Multiple cores sharing HW resources: L1 cache, Fetch+Decode Unit, (maybe even) Memory controller
  – GPU "Class" (NVIDIA Kepler, Maxwell, Parker..)
  – ~100s cores

✓ Multiple SMs integrated onto one chip
  – GPU "name" (NVIDIA GTX980, GT640…)
  – 1000s cores
  – NUMA hierarchy

✓ Typically (but not only) used as co-processor/accelerator
  – PCIEXPRESS connectivity
(GP)GPU programming stack
(GP)GPU programming stack

Application(s)

OpenGL

HW
(GP)GPU programming stack

Application(s)

Host

Device

OS hooks (APIs)

CUDA/OpenCL

API

Runtime

OS

HW

PCI

GPU programming stack

OpenGL

HW

HW

Host

Device

OS

Runtime

API

???
We need a programming model that provides

1. Simple offloading subroutines
2. An easy way to write code which runs on thousand threads
3. A way to exploit the NUMA hierarchy
1) Offload-based programming

✓ Offload-based programming models
  – CUDA
  – OpenCL
  – OpenMP 4.5
2) Parallelism in CUDA

✓ Exposed in the programming model
✓ Based on the concepts of
  ‒ Grid(s)
  ‒ Block(s)
  ‒ Thread(s)
2) Parallelism in CUDA

✓ Esposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)
  – Thread(s)
2) Parallelism in CUDA

✓ Esposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)
  – Thread(s)
2) Parallelism in CUDA

✓ Esposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)

```c
myKernel<<< /* NBLOCKS */ , 5 /* NTHREADS */ >>>();
```

Device

```
Grid #0
  Block (0,0)  Block (1,0)  Block (2,0)
  Block       Block       Block

Grid #1
  Block (0)   Block (1)   Block (2)
  Block (1,0) Block (1,1) Block (1,2)

Kernel #1

Kernel #N
```

```
Block (1,0)
  Thread (0)  Thread (1)  Thread (2)  Thread (3)  Thread (4)

Block (0,0)  Block (0,1)  Block (0,2)

Block (1,0)  Block (1,1)  Block (1,2)
```

```
Block (0,0)  Block (0,1)  Block (0,2)
  Block       Block       Block
```

```c
myKernel<<< /* NBLOCKS */ , 5 /* NTHREADS */ >>>();
```
2) Parallelism in CUDA

✓ Exposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)
  – Thread(s)
2) Parallelism in CUDA

✓ Exposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)
  – Thread(s)
2) Parallelism in CUDA

✓ Esposed in the programming model
✓ Based on the concepts of
  – Grid(s)
  – Block(s)
  – Thread(s)

```
dim3 grid_size;
grid_size.x = 3;
grid_size.y = 2;

dim3 blk_size;
blk_size.x = 5;
blk_size.y = 3;

myKernel<<<grid_size,blk_size>>>();
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  – Ultimately, the same Program Counter!!!
  – Threads cannot do branches - LOCKSTEP

```c
/* ... */

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  - Ultimately, the same Program Counter!!
  - Threads cannot do branches - LOCKSTEP

```
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

/* ... */
// l => # Blocks
// imgDim => #Threads
// l thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  – Ultimately, the same Program Counter!!!
  – Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

/* ... */
// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
  outputImg[thrId] = WHITE;
else
  outputImg[thrId] = BLACK;
/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  – Ultimately, the same Program Counter!!!
  – Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

int thrId = threadIdx.x;

/* ... */

// l => # Blocks
// imgDim => #Threads
// l thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
  outputImg[thrId] = WHITE;
else
  outputImg[thrId] = BLACK;

/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  – Ultimately, the same Program Counter!!!
  – Threads cannot do branches - LOCKSTEP

```
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

thrId 0

thrId 1

int thrId = threadIdx.x;
```

```c
/* ... */

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
   outputImg[thrId] = WHITE;
else
   outputImg[thrId] = BLACK;

/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
   – Ultimately, the same Program Counter!!!
   – Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;
```

/* ... */

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;
/* ... */
✓ (Groups of) cores share the same instruction Fetch/Decode Units
– Ultimately, the same Program Counter!!!
– Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
- Ultimately, the same Program Counter!!!
- Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;
/* ... */
```

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;
/* ... */
✓ (Groups of) cores share the same instruction Fetch/Decode Units
   – Ultimately, the same Program Counter!!!
   – Threads cannot do branches - LOCKSTEP

GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

thrId 0

thrId 1

```c
int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;
```

/* ... */

// l => # Blocks
// imgDim => #Threads
// l thread works on each pixel
int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
✓ (Groups of) cores share the same instruction Fetch/Decode Units

- Ultimately, the same Program Counter!!!
- Threads cannot do branches - LOCKSTEP

/* ... */

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
✓ (Groups of) cores share the same instruction Fetch/Decode Units
   – Ultimately, the same Program Counter!!!
   – Threads cannot do branches - LOCKSTEP

GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

thrId 0

thrId 1

int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    NOP

/* ... */

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  - Ultimately, the same Program Counter!!
  - Threads cannot do branches - LOCKSTEP

```c
GRAY_THRESHOLD = 150
inputImg[0] = 200
inputImg[1] = 100

int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
```

// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
```c
int thrId = threadIdx.x;
if (inputImg[thrId] >= GRAY_THRESHOLD)
    outputImg[thrId] = WHITE;
else
    outputImg[thrId] = BLACK;

/* ... */
```
✓ (Groups of) cores share the same instruction Fetch/Decode Units
  - Ultimately, the same Program Counter!!!
  - Threads cannot do branches - LOCKSTEP

```c
/* ... */
// 1 => # Blocks
// imgDim => #Threads
// 1 thread works on each pixel
int thrId = threadIdx.x;
if(inputImg[thrId] >= GRAY_THRESHOLD)
  outputImg[thrId] = WHITE;
  NOP
else
  NOP
  outputImg[thrId] = BLACK;
/* ... */
```
Warps, and lockstep

✓ Threads are grouped in warps
  – 1 warp <-> 32 CUDA threads
  – Units of scheduling
    – Threads of a single blocks are scheduled and de-scheduled 32 by 32
✓ Threads within the same warp run in LOCKSTEP
✓ Memory accesses within the single warp are coalesced
Integrated GP-GPUs

GP-GPU based embedded platforms
✓ As opposite to, traditional "discrete" GP-GPUs
✓ Still, host + accelerator model
✓ Communicate via shared memory
  – No PCI-express
  – CUDA "Unified Virtual Memory"
References

✓ "Calcolo parallelo" website
  – http://hipert.unimore.it/people/marko/courses/programmazione_parallela/

✓ My contacts
  – paolo.burgio@unimore.it
  – http://hipert.mat.unimore.it/people/paolob/

✓ Some pointers
  – https://www.khronos.org/
  – https://www.khronos.org/opencl/