Vrije Universiteit Brussel

ERASMUS HOGESCHOOL BRUSSEL

Computerarchitectuur

» Parallel Systems I
The GPU architecture

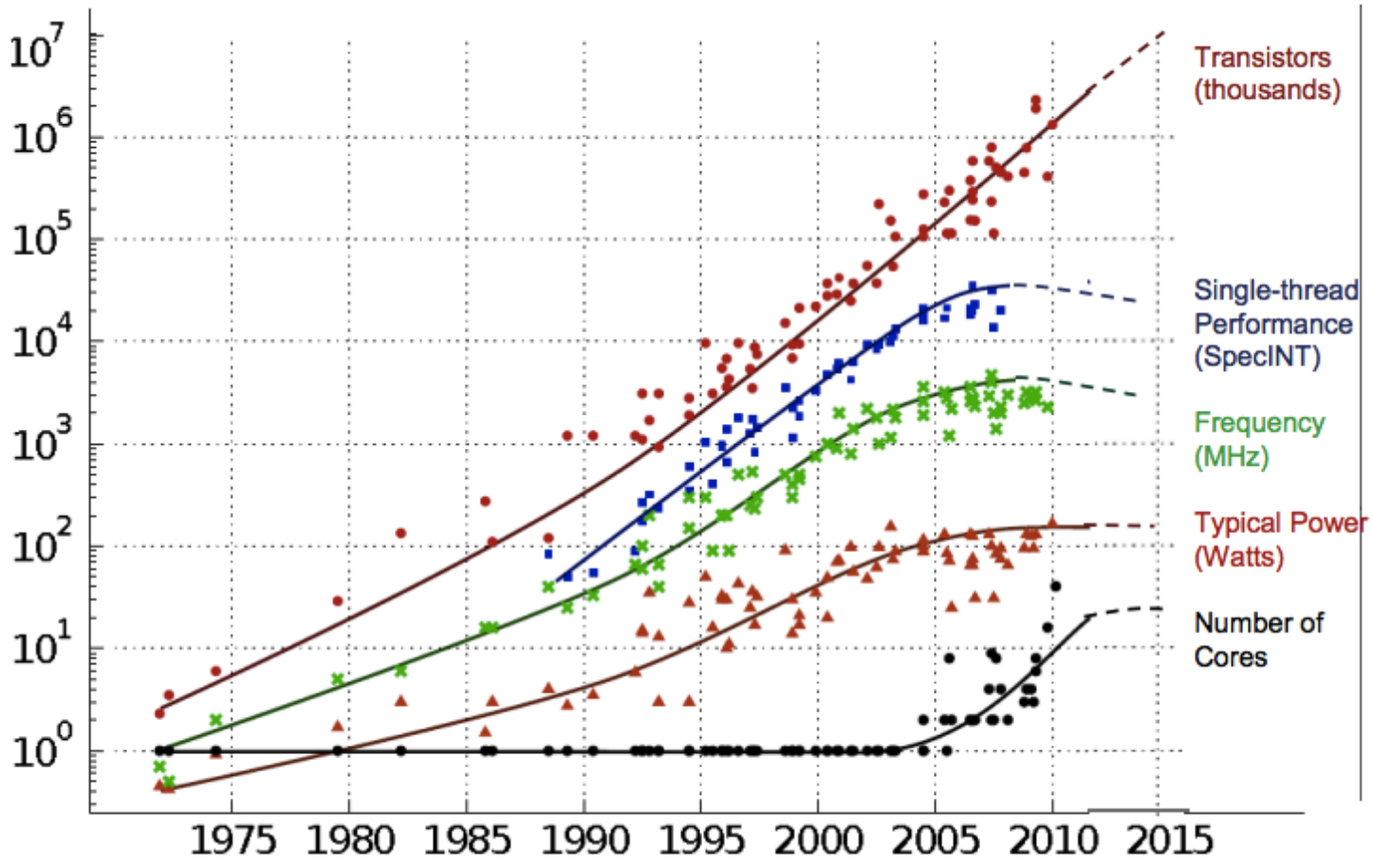
Jan Lemeire
2012-2013

CPU pipeline

- Sequential program
- 'Sequential' pipelined execution
- *Instruction-level parallelism (ILP):*
  - superscalar pipeline
  - out-of-order execution
- Limitations: max ILP = 4
  - Reached…
- Maximal clock frequency reached…
- Branch prediction, caching, forwarding, …
  - *More improvement possible?*

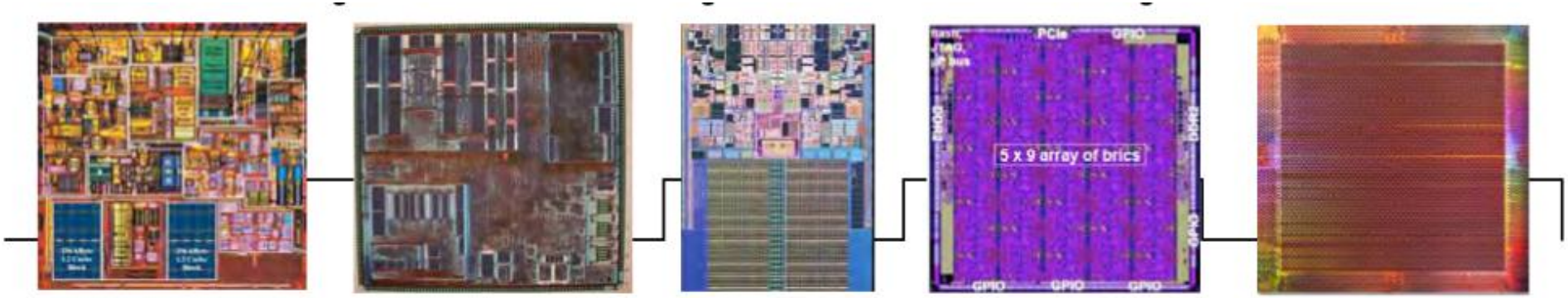
This is The End?

Way out:
parallel processors relying on explicit parallel software



Chuck Moore, "DATA PROCESSING IN EXASCALE-CLASS COMPUTER SYSTEMS", The Salishan Conference on High Speed Computing, 2011.

Parallel processors



CPUs | DSPs | Multicores | GPUs (arrays) | FPGAs

Single Cores

Multicores Coarse-Grained CPUs and DSPs

Coarse-Grained Massively Parallel Processor Arrays

Fine-Grained Massively Parallel Arrays

Courtesy of ALTERA

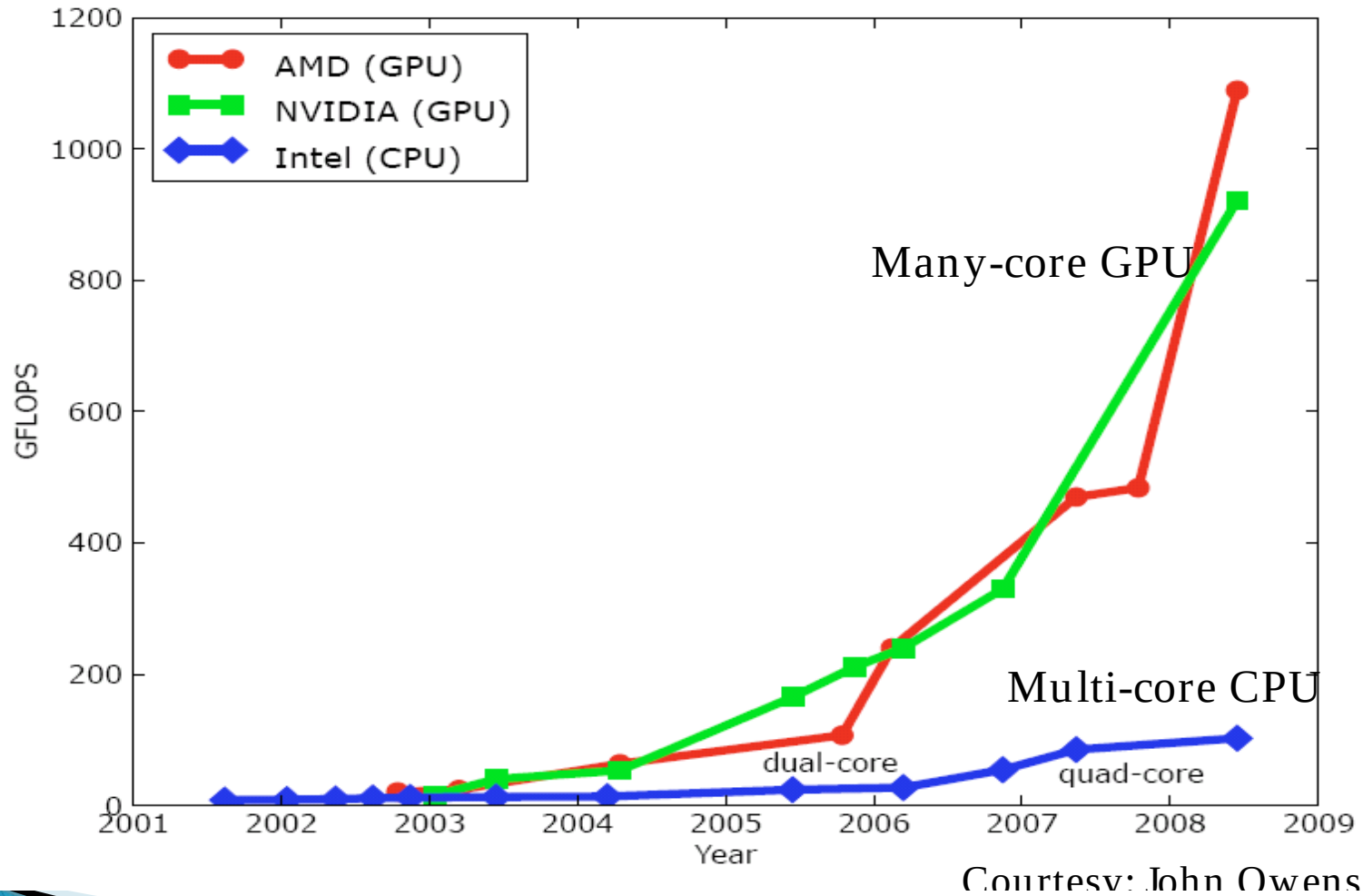
Thus:
The Future looks Parallel


Sequential world
Changed into…


Parallel world


2010
350 Million triangles/second
3 Billion transistors GPU
1995
5,000 triangles/second
800,000 transistors GPU
OGIO

Graphical Processing Units (GPUs)



Courtesy: John Owens

Supercomputing for free

- FASTRA at university of Antwerp

http://fastra.ua.ac.be

Collection of 8 graphical cards in PC

FASTRA 8 cards = 8x128 processors = 4000 euro

Similar performance as University's supercomputer (512 regular desktop PCs) that costed 3.5 million euro in 2005

Why are GPUs faster?

Devote transistors to… computation

GPU processor pipeline

- ±24 stages
- in-order execution!!
- no branch prediction!!
- no forwarding!!
- no register renaming!!
- Memory system:
  - relatively small
  - Until recently no caching
  - On the other hand: much more registers (see later)

New concepts:

1. Multithreading
2. SIMD

Multithreading on CPU

- 1 process/thread active per core
- When activating another thread: *context switch*

Overhead

  - Stop program execution: flush pipeline (let all instructions finish)
  - Save state of process/thread into **Process Control Block** : registers, program counter and operating system-specific data
  - Restore state of activated thread
  - Restart program execution and refill the pipeline

Hardware threads

- In several modern CPUs
  - typically 2HW threads
- Devote extra hardware for process state
- Context switch by hardware
  - (almost) no overhead
  - Within 1 cycle!
  - *Instructions in flight from different threads*

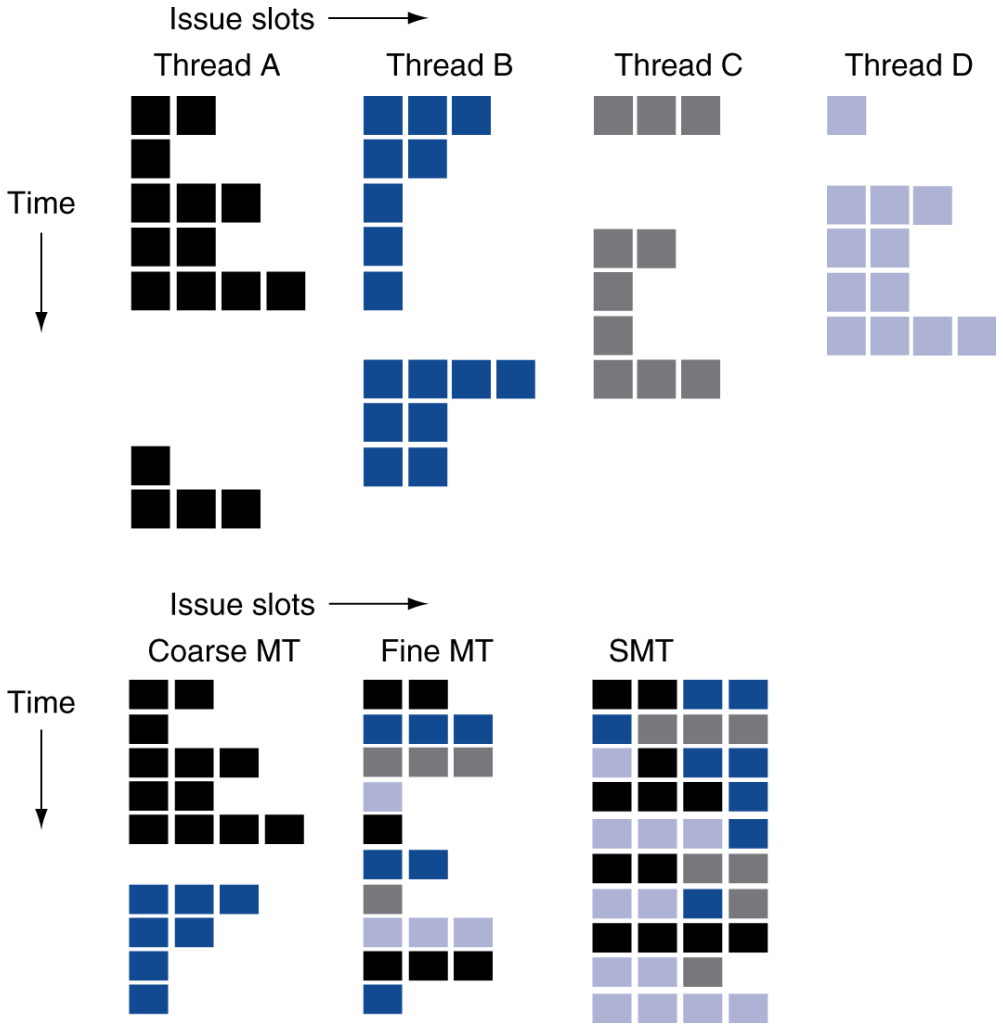
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example



Benefits of fine-grained multithreading

- Independent instructions (no bubbles)
- More time between instructions: possibility for *latency hiding*
  - Hide memory accesses
- If pipeline full
  - Forwarding not necessary
  - Branch prediction not necessary

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - lv, sv: load/store vector
    - addv.d: add vectors of double
    - addvs.d: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu r4,$s0,#512     ;upper bound of what to load
loop: l.d   $f2,0($s0)      ;load x(i)
      mul.d $f2,$f2,$f0     ;a × x(i)
      l.d   $f4,0($s1)      ;load y(i)
      add.d $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d   $f4,0($s1)      ;store into y(i)
      addiu $s0,$s0,#8      ;increment index to x
      addiu $s1,$s1,#8      ;increment index to y
      subu  $t0,r4,$s0      ;compute bound
      bne   $t0,$zero,loop  ;check if done
```

- Vector MIPS code

```
      l.d     $f0,a($sp)    ;load scalar a
      lv      $v1,0($s0)    ;load vector x
      mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
      lv      $v3,0($s1)    ;load vector y
      addv.d  $v4,$v2,$v3   ;add y to product
      sv      $v4,0($s1)    ;store the result
```

Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

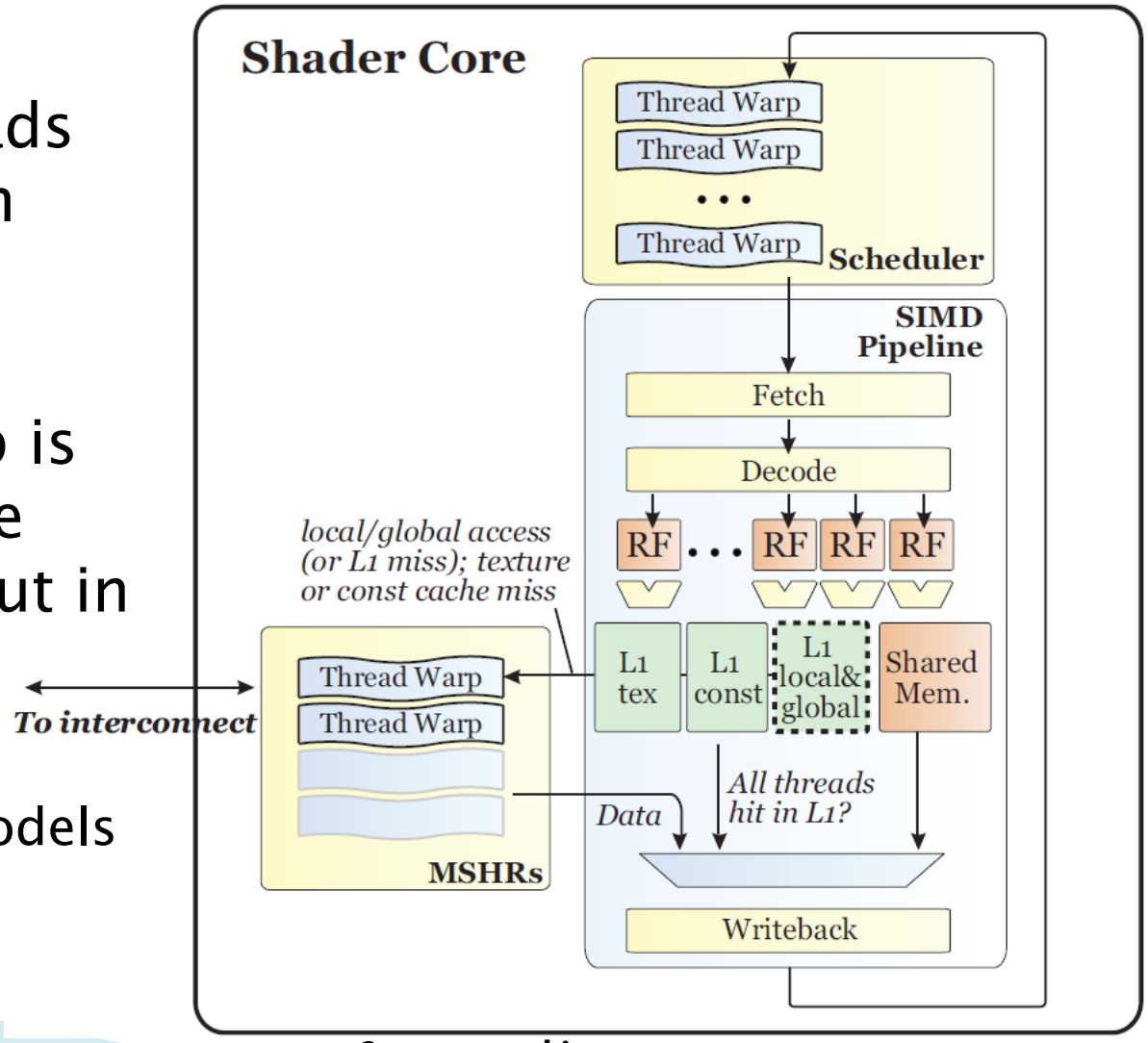
Now:
the GPU architecture

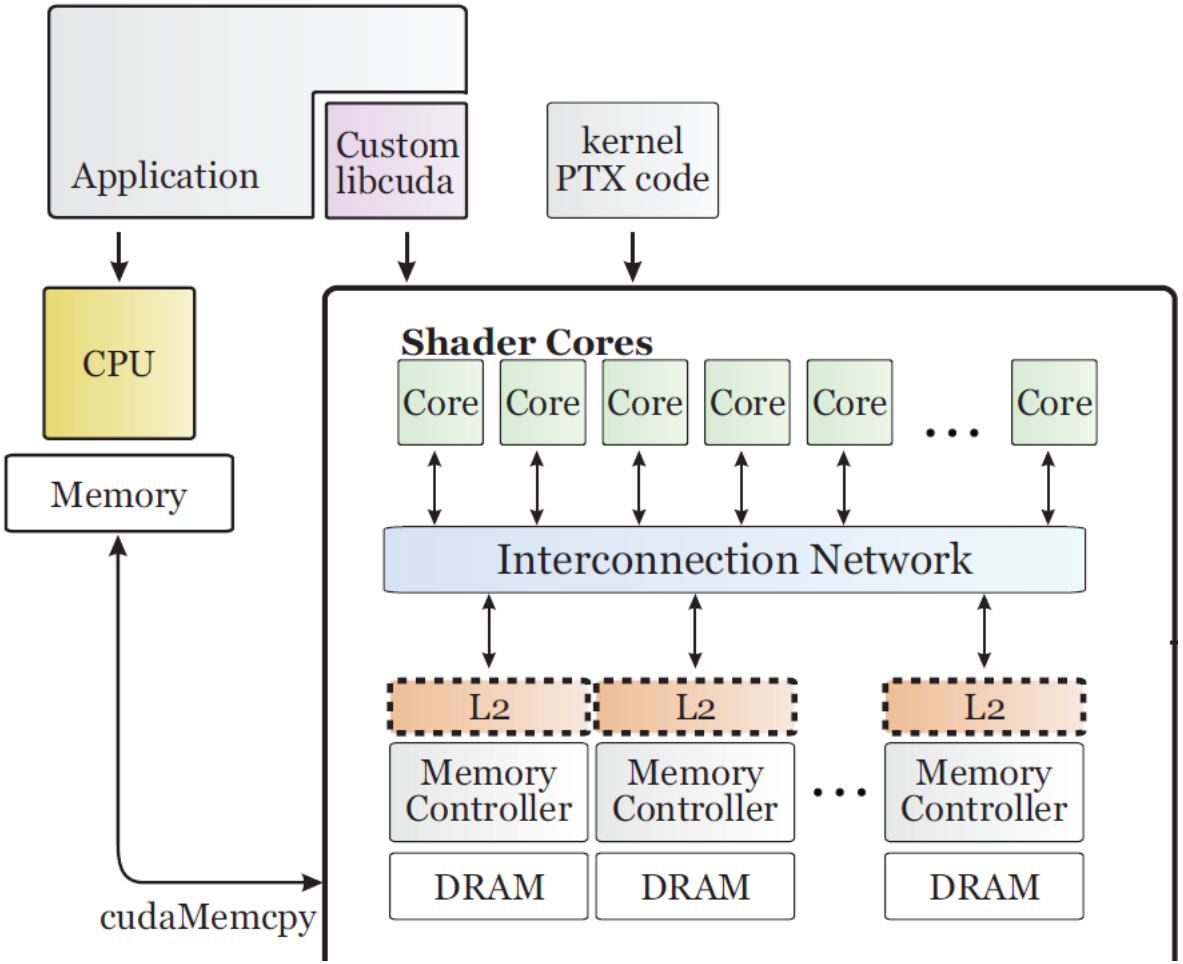
1 Streaming Multiprocessor

hardware threads grouped by 32 threads (warps), executing in lockstep (*SIMD*)

In each cycle, a warp is selected of which the next instruction is put in the pipeline

Ps: cache only in new models




Application
Custom libcuda
kernel PTX code
CPU
Memory
cudaMemcpy
Shader Cores
Core
Core
Core
Core
Core
Core
Interconnection Network
L2
L2
L2
Memory Controller
Memory Controller
Memory Controller
DRAM
DRAM
DRAM

Peak GPU Performance

- GPUs consist of Streaming MultiProcessors (MPs) grouping a number of Scalar Processors (SPs)
- Nvidia GTX 280:
  - 30MPs x 8 SPs/MP x 2FLOPs/instr/SP x 1 instr/clock x 1.3 GHz
  = **624 GFlops**
- Nvidia Tesla C2050:
  - 14 MPs x 32 SPs/MP x 2FLOPs/instr/SP x 1 instr/clock x 1.15 GHz (clocks per second)
  = **1030 GFlops**

Example: pixel transformation

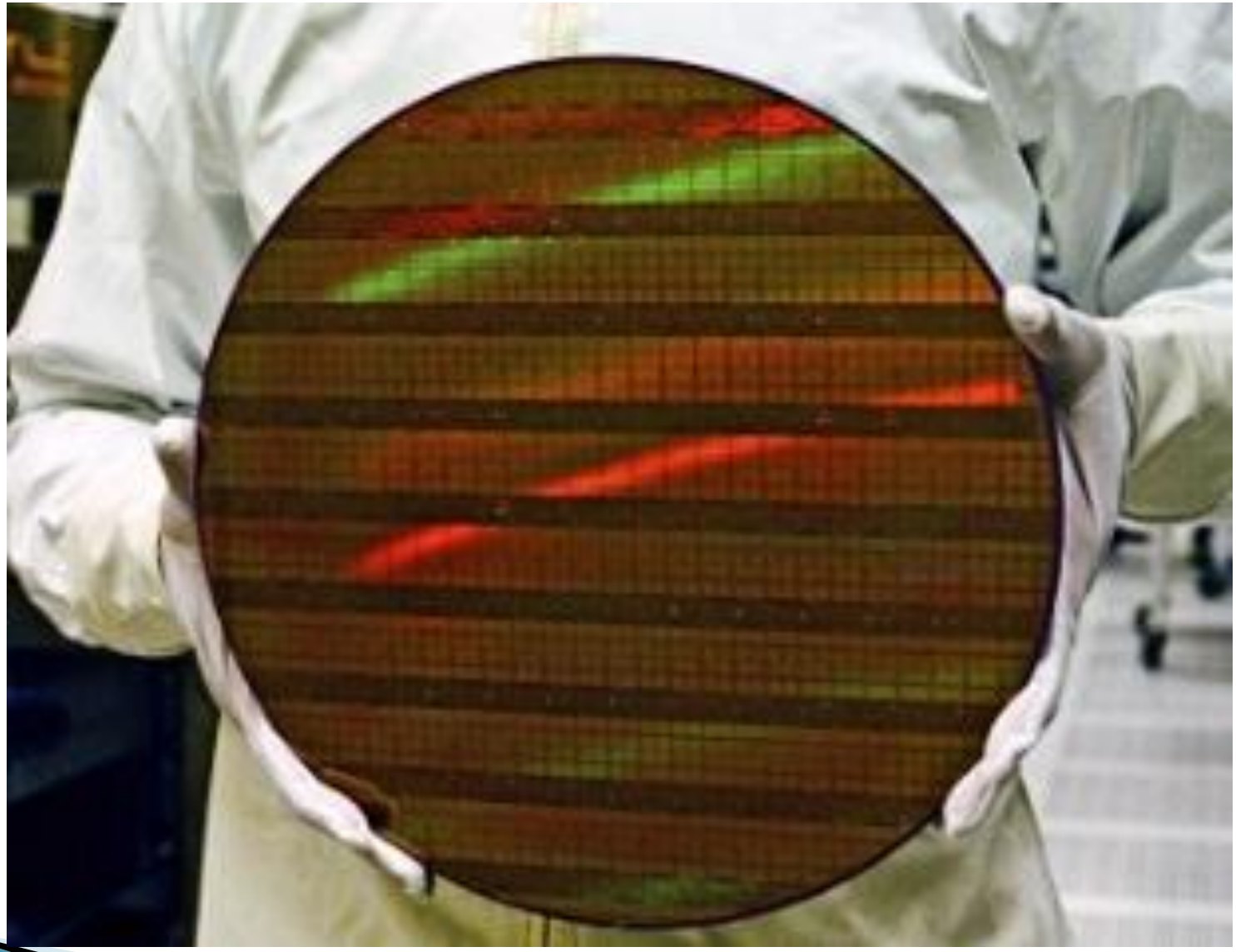
- Kernel executed by each thread, each processing 1 pixel

GPU = Zero-overhead thread processor

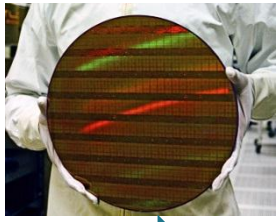
```
usgn_8  transform(usgn_8 in, sgn_16 gain, sgn_16 gain_divide,
sgn_8 offset)
{
    sgn_32 x;

    x = (in * gain / gain_divide) + offset;

    if (x < 0) x = 0;
    if (x > 255) x = 255;
    return x;
 }
```

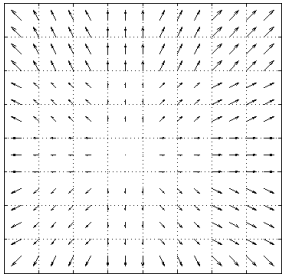


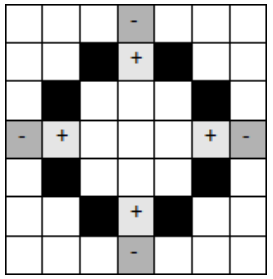
Example: real-time image processing



KLA-Tencor
Accelerating Yield

Images of 20MegaPixels





Pixel rescaling ***lens correction*** ***pattern detection***

CPU gives only 4 fps
next generation machines need 50fps

CPU: 4 fps  GPU: 70 fps

» However:
processing power not for free

Obstacle 1

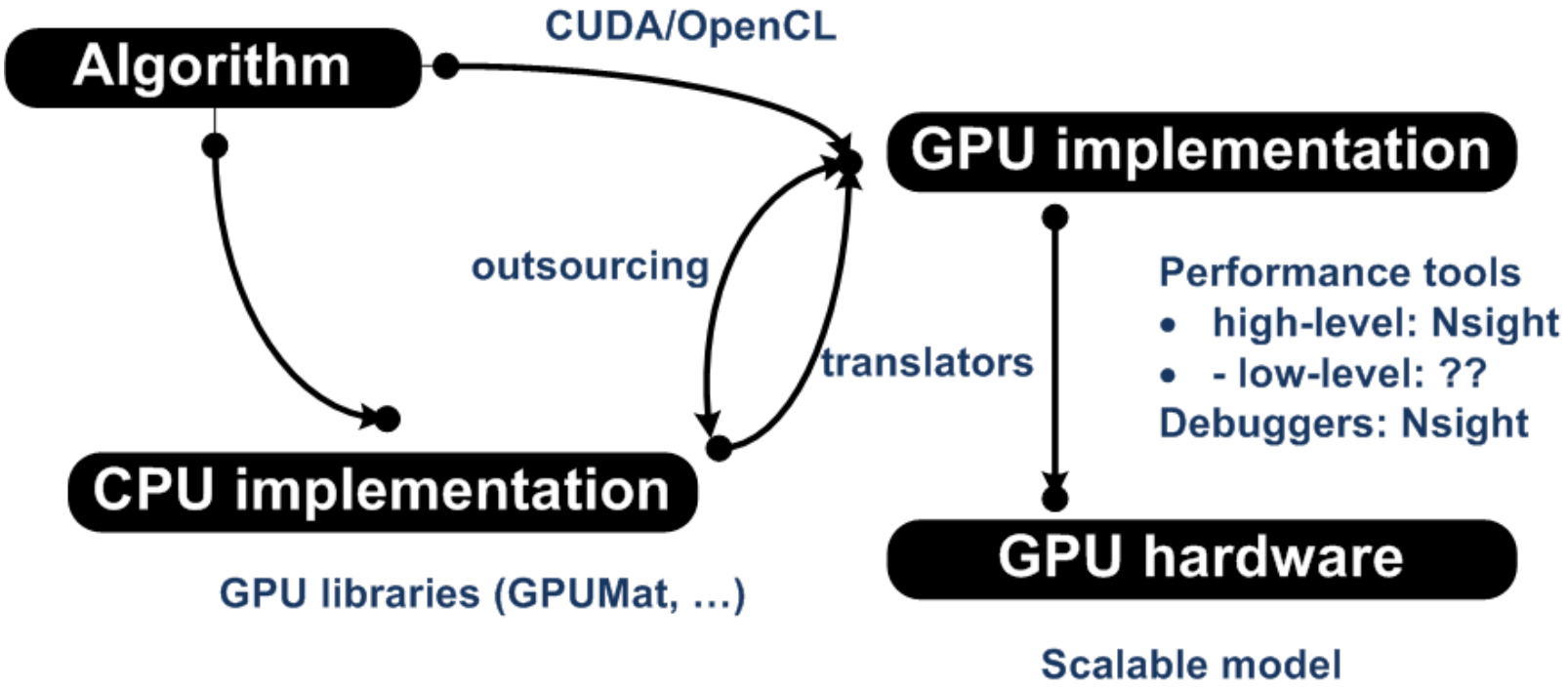
Hard(er) to implement

Obstacle 2

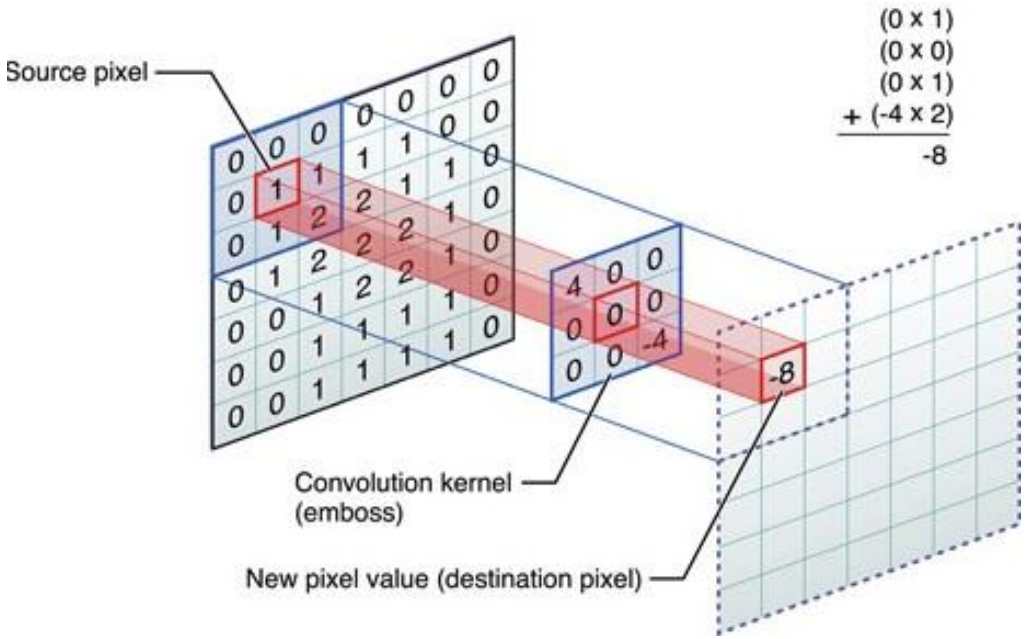
Hard(er) to get efficiency

X86 to GPU

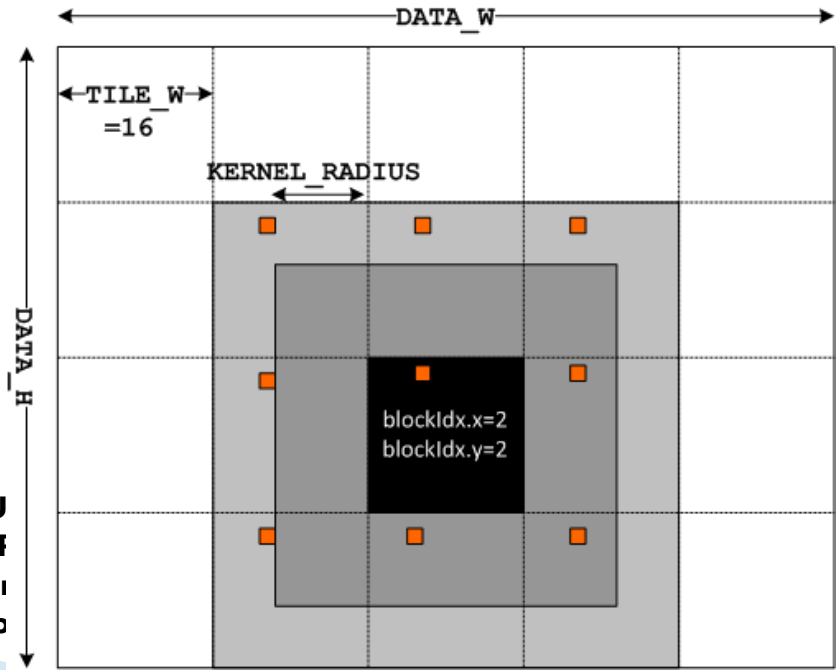
- Libraries (CUBLAS, CUFFT, …)
- Ease: THRUST, ...



Example 1: convolution



Parallelism: +++
Locality: ++
Work/pixel: ++



GU
GPU/FI
for
pro