Parallel programming: Introduction to GPU architecture

Caroline Collange
Inria Rennes – Bretagne Atlantique

caroline.collange@inria.fr
https://team.inria.fr/pacap/members/collange/

Master 1
PPAR - 2020
Outline of the course

- March 3: Introduction to GPU architecture
  - Parallelism and how to exploit it
  - Performance models
- March 9: GPU programming
  - The software side
  - Programming model
- March 16: Performance optimization
  - Possible bottlenecks
  - Common optimization techniques
- 4 lab sessions, starting March 17
  - Labs 1&2: computing log(2) the hard way
  - Labs 3&4: yet another Conway's Game of Life
Graphics processing unit (GPU)

- Graphics rendering accelerator for computer games
  - Mass market: low unit price, amortized R&D
  - Increasing programmability and flexibility
- Inexpensive, high-performance parallel processor
  - GPUs are everywhere, from cell phones to supercomputers
- General-Purpose computation on GPU (GPGPU)
GPUs in high-performance computing

- GPU/accelerator share in Top500 supercomputers
  - In 2010: 2%
  - In 2018: 22%

- 2016+ trend:
  Heterogeneous multi-core processors influenced by GPUs

#1 Summit (USA)
4,608 × (2 Power9 CPUs + 6 Volta GPUs)

#3 Sunway TaihuLight (China)
40,960 × SW26010 (4 big + 256 small cores)
GPUs in the future?

- **Yesterday (2000-2010)**
  - Homogeneous multi-core
  - Discrete components

- **Today (2011-...)**
  - Chip-level integration
    - CPU cores and GPU cores on the same chip
    - Still different programming models, software stacks

- **Tomorrow**
  - Heterogeneous multi-core
    - GPUs to blend into throughput-optimized, general purpose cores?
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
  - Hardware trends

- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical

- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
The free lunch era... was yesterday

- 1980's to 2002: *Moore's law, Dennard scaling*, micro-architecture improvements
  - Exponential performance increase
  - Software compatibility preserved

Hennessy, Patterson. Computer Architecture, a quantitative approach. 5th Ed. 2010

- Do not rewrite software, buy a new machine!
Technology evolution

- **Memory wall**
  - Memory speed does not increase as fast as computing speed
  - Harder to hide memory latency

- **Power wall**
  - Power consumption of transistors does not decrease as fast as density increases
  - Performance is now limited by power consumption

- **ILP wall**
  - Law of diminishing returns on Instruction-Level Parallelism
  - Pollack rule: cost \( \approx \) performance^2
Usage changes

- New applications demand **parallel processing**
  - Computer games: 3D graphics
  - Search engines, social networks... “big data” processing

- New computing devices are **power-constrained**
  - Laptops, cell phones, tablets...
    - Small, light, battery-powered
  - Datacenters
    - High power supply and cooling costs
Latency vs. throughput

- **Latency**: time to solution
  - Minimize time, at the expense of power
  - Metric: time
e.g. seconds
- **Throughput**: quantity of tasks processed per unit of time
  - Assumes unlimited parallelism
  - Minimize energy per operation
  - Metric: operations / time
e.g. Gflops / s
- CPU: optimized for latency
- GPU: optimized for throughput
Amdahl's law

- Bounds speedup attainable on a parallel machine

\[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]

- Time to run sequential portions
- Time to run parallel portions

S (speedup)

N (available processors)

Why heterogeneous architectures?

\[ S = \frac{1}{(1-P) + \frac{P}{N}} \]

- Latency-optimized multi-core (CPU)
  - Low efficiency on parallel portions: spends too much resources

- Throughput-optimized multi-core (GPU)
  - Low performance on sequential portions

- Heterogeneous multi-core (CPU+GPU)
  - Use the right tool for the right job
  - Allows aggressive optimization for latency \textbf{or} for throughput

Example: System on Chip for smartphone

- Big cores for applications
- Tiny cores for background activity
- Lots of interfaces
- Special-purpose accelerators
- GPU
- 3G/4G modem
- WiLink™ wireless connectivity
- OMAP5430
- Dynamic memory manager
  - L2 cache
  - ARM Cortex-M4
  - ARM Cortex-M4
- Dynamic Network-on-chip interconnect
  - L3 Network-on-chip interconnect
  - L4 peripherals
  - L4 peripherals
- M-Shield™ system security technology: SHA-1, SHA-2, MD5, DES,3DES, RNG, AES, PKA, secure WDT, keys, crypto DMA
- Multi-flite display sub-system (DSS)
- Audio processor
- Timer, Interrupt Controller, Mailboxes, System DMA
- Boot/Secure ROM, L3 RAM
- ARM® Cortex™-A15 MPCore (up to 2 GHz)
- ARM Cortex-A15 MPCore (up to 2 GHz)
- PowerVR™ SGX544-MPX 3D graphics
- TI 2D graphics
- C64x IVA-HD video accelerator
- Up to four cameras
- Camera control
- DIG MIC
- 32 kHz Crystal
- In/Out
- TWL6040
- Audio
- Headset
- Speakers
- Vibrators
- Amplifiers
- Micro
- LCD
- MIPI DSI
- MIPI DSI
- MIPI DBI-B/DPI
- HDMI 1.4a
- Up to four displays
- Emulator pod, Trace analyzer
- UART, GPIO, Keypad
- Debug & trace, cJTAG/STP/PTM
- Fast IrDA
- EMMC
- SD
- USB SS/HS host/target
- USB HS host
- Companion device
- MIPI LLC/UniPort™-M
- MIPI CSI-3
- MIPI CSI-2 + CPI
- SMP (up to 2 GHz)
- UART/SPI
- SDIO
- McBSP
- FC/SPI
- SDIO
- UART
- McBSP
- HDQ/1-Wire
- REF/CLK
- UARTs
- Debug & trace, cJTAG/STP/PTM
- UART, GPIO, Keypad
- Touch screen controller
- HDMI 1.4a
- 3D HDTV
- LCD
- MIPI DSI
- MIPI DSI
- MIPI DBI-B/DPI
- HDMI 1.4a
- Up to four displays
- Emulator pod, Trace analyzer
- UART, GPIO, Keypad
- Debug & trace, cJTAG/STP/PTM
- Fast IrDA
- EMMC
- SD
- USB SS/HS host/target
- USB HS host
- Companion device
- MIPI LLC/UniPort™-M
- MIPI CSI-3
- MIPI CSI-2 + CPI
- SMP (up to 2 GHz)
- UART/SPI
- SDIO
- McBSP
- FC/SPI
- SDIO
- UART
- McBSP
- HDQ/1-Wire
- REF/CLK
- UARTs
- Debug & trace, cJTAG/STP/PTM
- UART, GPIO, Keypad
- Touch screen controller
- HDMI 1.4a
- 3D HDTV
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
  - Hardware trends

- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical

- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
The (simplest) graphics rendering pipeline

1. Primitives (triangles…)
2. Vertices
3. Vertex shader
4. Clipping, Rasterization, Attribute interpolation
5. Fragments
6. Fragment shader
7. Z-Compare
8. Blending
9. Framebuffer
10. Textures
11. Pixels
12. Parametrizable stage
13. Programmable stage
How much performance do we need...

- to run 3DMark 11 at 50 frames/second?

<table>
<thead>
<tr>
<th>Element</th>
<th>Per frame</th>
<th>Per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>12.0M</td>
<td>600M</td>
</tr>
<tr>
<td>Primitives</td>
<td>12.6M</td>
<td>630M</td>
</tr>
<tr>
<td>Fragments</td>
<td>180M</td>
<td>9.0G</td>
</tr>
<tr>
<td>Instructions</td>
<td>14.4G</td>
<td>720G</td>
</tr>
</tbody>
</table>

Intel Core i7 2700K: 56 Gips/s peak
- We need to go 13x faster
- Make a special-purpose accelerator

Source: Damien Triolet, Hardware.fr
# GPGPU: General-Purpose computation on GPUs

## GPGPU history summary

### Microsoft DirectX

<table>
<thead>
<tr>
<th>Year</th>
<th>DirectX Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>7.x</td>
</tr>
<tr>
<td>2001</td>
<td>8.0</td>
</tr>
<tr>
<td>2002</td>
<td>8.1</td>
</tr>
<tr>
<td>2003</td>
<td>9.0 a</td>
</tr>
<tr>
<td>2004</td>
<td>9.0b</td>
</tr>
<tr>
<td>2005</td>
<td>9.0c</td>
</tr>
<tr>
<td>2006</td>
<td>10.0</td>
</tr>
<tr>
<td>2007</td>
<td>10.1</td>
</tr>
<tr>
<td>2008</td>
<td>11</td>
</tr>
</tbody>
</table>

**Unified shaders**

### NVIDIA

<table>
<thead>
<tr>
<th>Year</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>NV10</td>
</tr>
<tr>
<td>2001</td>
<td>NV20</td>
</tr>
<tr>
<td>2002</td>
<td>NV30</td>
</tr>
<tr>
<td>2003</td>
<td>NV40</td>
</tr>
<tr>
<td>2004</td>
<td>G70</td>
</tr>
<tr>
<td>2005</td>
<td>G80-G90</td>
</tr>
<tr>
<td>2006</td>
<td>GT200</td>
</tr>
<tr>
<td>2007</td>
<td>GF100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP 16</td>
<td>NV10</td>
</tr>
<tr>
<td>Programmable shaders</td>
<td>NV20</td>
</tr>
<tr>
<td>FP 32</td>
<td>NV30</td>
</tr>
<tr>
<td>Dynamic control flow</td>
<td>NV40</td>
</tr>
<tr>
<td>SIMT</td>
<td>G70</td>
</tr>
<tr>
<td>CUDA</td>
<td></td>
</tr>
</tbody>
</table>

### ATI/AMD

<table>
<thead>
<tr>
<th>Year</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>R100</td>
</tr>
<tr>
<td>2001</td>
<td>R200</td>
</tr>
<tr>
<td>2002</td>
<td>R300</td>
</tr>
<tr>
<td>2003</td>
<td>R400</td>
</tr>
<tr>
<td>2004</td>
<td>R500</td>
</tr>
<tr>
<td>2005</td>
<td>R600</td>
</tr>
<tr>
<td>2006</td>
<td>R700</td>
</tr>
<tr>
<td>2007</td>
<td>Evergreen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP 24</td>
<td>R100</td>
</tr>
<tr>
<td>CTM</td>
<td>R200</td>
</tr>
<tr>
<td>FP 64</td>
<td>R300</td>
</tr>
<tr>
<td>CAL</td>
<td>R400</td>
</tr>
</tbody>
</table>

---

### GPGPU traction

- **2000**
- **2001**
- **2002**
- **2003**
- **2004**
- **2005**
- **2006**
- **2007**
- **2008**
- **2009**
- **2010**

---

*Note: This summary includes key milestones in the evolution of GPGPU technology, focusing on the development of shader architectures and the introduction of new programming models.*
Today: what do we need GPUs for?

1. 3D graphics rendering for games
   - Complex texture mapping, lighting computations…
2. Computer Aided Design workstations
   - Complex geometry
3. High-performance computing
   - Complex synchronization, off-chip data movement, high precision
4. Convolutional neural networks
   - Complex scheduling of low-precision linear algebra

One chip to rule them all
   - Find the common denominator
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
  - Hardware trends

- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical

- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
Caveat: only considers desktop CPUs. Gap with server CPUs is “only” 4×!
Trends: memory bandwidth

- HBM, GDDR5x, 6x
- Integrated mem. controller

Year


Peak DRAM bandwidth (GB/s)

8800 GTX 2900 XT 9800 3870 4870 GTX 486 GTX 488 GTX 480 GTX 5890 GTX 788 K20x R9 290X GTX 980 R9 Fury X Titan X RX Vega 64 RTX 2080 Ti

10x

Integrated mem. controller

AMD GPU

Intel CPU

Intel MIC

HBM, GDDR5x, 6x
Trends: energy efficiency
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
  - Hardware trends

- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical

- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
What is parallelism?

Parallelism: independent operations which execution can be overlapped
Operations: memory accesses or computations

How much parallelism do I need?
- Little's law in queuing theory
  - Average customer arrival rate $\lambda$
  - Average time spent $W$ ← throughput
  - Average number of customers \( L = \lambda \times W \)
  ← latency
  ← Parallelism = throughput $\times$ latency

Units
- For memory: \( B = GB/s \times ns \)
- For arithmetic: \( \text{flops} = G\text{flops/s} \times ns \)

J. Little. A proof for the queuing formula $L = \lambda W$. JSTOR 1961.
Throughput and latency: CPU vs. GPU

CPU memory: Core i7 4790, DDR3-1600, 2 channels

Throughput (GB/s) 25.6
Latency (ns) 67

GPU memory: NVIDIA GeForce GTX 980, GDDR5-7010, 256-bit

Throughput x8
Parallelism: x56
Latency x6

→ Need 56 times more parallelism!
Consequence: more parallelism

- GPU vs. CPU
  - 8× more parallelism to feed more units (throughput)
  - 6× more parallelism to hide longer latency
  - 56× more total parallelism
- How to find this parallelism?
Sources of parallelism

- **ILP: Instruction-Level Parallelism**
  - Between independent instructions in sequential program

- **TLP: Thread-Level Parallelism**
  - Between independent execution contexts: threads

- **DLP: Data-Level Parallelism**
  - Between elements of a vector: same operation on several elements

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\text{Parallel} & \\
\text{add} & \quad \text{add} \quad \text{mul} \\
\text{add} & \quad \text{add} \quad \text{mul} \\
\text{mul} & \quad \text{mul} \\
\text{sub} & \quad \text{sub} \\
\end{align*}
\]

\[
\begin{align*}
vadd \quad r & \leftarrow a, b \\
& \quad a_1 \quad a_2 \quad a_3 \\
& \quad + \quad + \quad + \\
& \quad b_1 \quad b_2 \quad b_3 \\
& \quad r_1 \quad r_2 \quad r_3
\end{align*}
\]
Example: $X ← a \times X$

- In-place scalar-vector product: $X ← a \times X$

  **Sequential (ILP)**
  
  For $i = 0$ to $n-1$ do:
  
  $X[i] ← a \times X[i]$

  **Threads (TLP)**
  
  Launch $n$ threads:
  
  $X[tid] ← a \times X[tid]$

  **Vector (DLP)**
  
  $X ← a \times X$

- Or any combination of the above
Uses of parallelism

- "Horizontal" parallelism for throughput
  - More units working in parallel

- "Vertical" parallelism for latency hiding
  - Pipelining: keep units busy when waiting for dependencies, memory

![Diagram showing horizontal and vertical parallelism]
How to extract parallelism?

<table>
<thead>
<tr>
<th></th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>Superscalar</td>
<td>Pipelined</td>
</tr>
<tr>
<td>TLP</td>
<td>Multi-core SMT</td>
<td>Interleaved / switch-on-event multithreading</td>
</tr>
<tr>
<td>DLP</td>
<td>SIMD / SIMT</td>
<td>Vector / temporal SIMT</td>
</tr>
</tbody>
</table>

- We have seen the first row: ILP
- We will now review techniques for the next rows: TLP, DLP
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
  - Hardware trends
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
Sequential processor

- Focuses on instruction-level parallelism
  - Exploits ILP: vertically (pipelining) and horizontally (superscalar)

\[
\text{for } i = 0 \text{ to } n-1 \\
X[i] \leftarrow a \times X[i]
\]

Source code

\[
\text{move } i \leftarrow 0 \\
\text{loop:} \\
\text{load } t \leftarrow X[i] \\
\text{mul } t \leftarrow a \times t \\
\text{store } X[i] \leftarrow t \\
\text{add } i \leftarrow i+1 \\
\text{branch } i < n? \text{ loop}
\]

Machine code

- Fetch
- Decode
- Execute
- Memory

Sequential CPU
The incremental approach: multi-core

- Several processors on a single chip sharing one memory space

- Area: benefits from Moore's law
- Power: extra cores consume little when not in use
  - e.g. Intel Turbo Boost
Homogeneous multi-core

- Horizontal use of thread-level parallelism

- Improves peak throughput
Example: Tilera Tile-GX

- Grid of (up to) 72 tiles
- Each tile: 3-way VLIW processor, 5 pipeline stages, 1.2 GHz
Interleaved multi-threading

- Vertical use of thread-level parallelism

Hides latency thanks to explicit parallelism improves achieved throughput
Example: Oracle Sparc T5

- 16 cores / chip
- Core: out-of-order superscalar, 8 threads
- 15 pipeline stages, 3.6 GHz
Clustered multi-core

- For each individual unit, select between
  - Horizontal replication
  - Vertical time-multiplexing

  **Examples**
  - Sun UltraSparc T2, T3
  - AMD Bulldozer
  - IBM Power 7, 8, 9

- Area-efficient tradeoff
- Blurs boundaries between cores
Implicit SIMD

- **Factorization** of fetch/decode, load-store units
  - Fetch 1 instruction on behalf of several threads
  - Read 1 memory location and broadcast to several registers

- In NVIDIA-speak
  - SIMT: Single Instruction, Multiple Threads
  - Convoy of synchronized threads: *warp*

- Extracts DLP from multi-thread applications
How to exploit common operations?

Multi-threading implementation options:

- Horizontal: replication
  - Different resources, same time
  - Chip Multi-Processing (CMP)
- Vertical: time-multiplexing
  - Same resource, different times
  - Multi-Threading (MT)
- Factorization
  - If we have common operations between threads
  - Same resource, same time
  - Single-Instruction Multi-Threading (SIMT)
Explicit SIMD

- Single Instruction Multiple Data
- Horizontal use of data level parallelism

Examples
- Intel MIC (16-wide)
- AMD GCN GPU (16-wide×4-deep)
- Most general purpose CPUs (4-wide to 16-wide)
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
Example CPU: Intel Core i7

- Is a wide superscalar, but has also
  - Multicore
  - Multi-thread / core
  - SIMD units
- Up to 116 operations/cycle from 8 threads
Example GPU: NVIDIA GeForce GTX 980

- SIMT: warps of 32 threads
- 16 SMs / chip
- $4 \times 32$ cores / SM, 64 warps / SM

- 4612 Gflop/s
- Up to 32768 threads in flight
## Taxonomy of parallel architectures

<table>
<thead>
<tr>
<th></th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>Superscalar / VLIW</td>
<td>Pipelined</td>
</tr>
<tr>
<td>TLP</td>
<td>Multi-core SMT</td>
<td>Interleaved / switch-on-event multithreading</td>
</tr>
<tr>
<td>DLP</td>
<td>SIMD / SIMT</td>
<td>Vector / temporal SIMT</td>
</tr>
</tbody>
</table>
### Classification: multi-core

<table>
<thead>
<tr>
<th></th>
<th>Intel Haswell</th>
<th>Fujitsu SPARC64 X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Horizontal</strong></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Vertical</strong></td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td><strong>Cores</strong></td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td><strong>Hyperthreading</strong></td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td><strong>SIMD (AVX)</strong></td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

**General-purpose multi-cores:** balance ILP, TLP and DLP

**IBM Power 8**

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>12</th>
<th>8</th>
</tr>
</thead>
</table>

**Oracle Sparc T5**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
</table>

**Sparc T:** focus on TLP
How to read the table

- Given an application with known ILP, TLP, DLP, how much throughput / latency hiding can I expect?
- For each cell, take minimum of existing parallelism and hardware capability
- The column-wise product gives throughput / latency hiding

<table>
<thead>
<tr>
<th>Sequential code</th>
<th>Horizontal</th>
<th>Vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>10</td>
<td>min(8, 10) = 8</td>
</tr>
<tr>
<td>TLP</td>
<td>1</td>
<td>min(4, 1) = 1</td>
</tr>
<tr>
<td>DLP</td>
<td>1</td>
<td>min(8, 1) = 1</td>
</tr>
</tbody>
</table>

Max throughput = 8×1×1 for this application
Peak throughput = 8×4×8 that can be achieved
→ Can only hope for ~3% of peak performance!
Classification: GPU and many small-core

<table>
<thead>
<tr>
<th></th>
<th>Intel MIC</th>
<th>Nvidia Kepler</th>
<th>AMD GCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal</td>
<td>ILP: 2</td>
<td>TLP: 2</td>
<td>AMD: 20×4</td>
</tr>
<tr>
<td></td>
<td>TLP: 60</td>
<td>Vertical: 4</td>
<td>Vertical: 40</td>
</tr>
<tr>
<td></td>
<td>DLP: 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIMD: 2</td>
<td>Cores: 16×4</td>
<td>Multi-threading: 40</td>
</tr>
<tr>
<td></td>
<td>Cores: 16</td>
<td>SIMT: 32</td>
<td></td>
</tr>
</tbody>
</table>

GPU: focus on DLP, TLP horizontal and vertical

Many small-core: focus on horizontal TLP

Tilera Tile-GX

Kalray MPPA-256
Takeaway

- Parallelism for throughput and latency hiding
- Types of parallelism: ILP, TLP, DLP
- All modern processors exploit the 3 kinds of parallelism
- GPUs focus on Thread-level and Data-level parallelism
Outline

- GPU, many-core: why, what for?
  - Technological trends and constraints
  - From graphics to general purpose
- Forms of parallelism, how to exploit them
  - Why we need (so much) parallelism: latency and throughput
  - Sources of parallelism: ILP, TLP, DLP
  - Uses of parallelism: horizontal, vertical
- Let's design a GPU!
  - Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
  - Putting it all together
  - Architecture of current GPUs: cores, memory
What is inside a graphics card?

NVIDIA Volta V100 GPU. Artist rendering!
External memory: discrete GPU

Classical CPU-GPU model

- Split memory spaces
- Need to transfer data explicitly
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

Example configuration:
Intel Core i7 4790, Nvidia GeForce GTX 980
Discrete GPU memory technology

- **GDDR5, GDDR5x**
  - Qualitatively like regular DDR
  - Optimized for high frequency at the expense of latency and cost
  - e.g. *Nvidia Titan X*: 12 chip pairs x 32-bit bus x 10 GHz → 480 GB/s

- **High-Bandwidth Memory (HBM)**
  - On-package stacked memory on silicon interposer
  - Shorter trace, wider bus, lower frequency: more energy-efficient
  - Limited capacity and high cost
  - e.g. *AMD R9 Fury X*: 4× 4-high stack x 1024-bit x 1 GHz → 512 GB/s
Most GPUs today are integrated
- Same physical memory
- May support memory coherence
  - GPU can read directly from CPU caches
- More contention on external memory
GPU high-level organization

- **Processing units**
  - Streaming Multiprocessors (SM) in Nvidia jargon
  - Compute Unit (CU) in AMD's
  - Closest equivalent to a CPU core
  - Today: from 1 to 20 SMs in a GPU

- **Memory system: caches**
  - Keep frequently-accessed data
  - Reduce throughput demand on main memory
  - Managed by hardware (L1, L2) or software (Shared Memory)
Each SM is a highly-multithreaded processor

- Today: 24 to 48 warps of 32 threads each
  → ~1K threads on each SM, ~10K threads on a GPU
GPU, many-core: why, what for?
- Technological trends and constraints
- From graphics to general purpose

Forms of parallelism, how to exploit them
- Why we need (so much) parallelism: latency and throughput
- Sources of parallelism: ILP, TLP, DLP
- Uses of parallelism: horizontal, vertical

Let's design a GPU!
- Ingredients: Sequential core, Multi-core, Multi-threaded core, SIMD
- Putting it all together
- Architecture of current GPUs: cores, memory

High-level performance modeling
First-order performance model

Questions you should ask yourself, before starting to code or optimize

- Will my code run faster on the GPU?
- Is my existing code running as fast as it should?
- Is performance limited by computations or memory bandwidth?

Pen-and-pencil calculations can (often) answer such questions
Performance: metrics and definitions

Optimistic evaluation: upper bound on performance
Assume perfect overlap of computations and memory accesses

- **Memory accesses**: bytes
  - Only external memory, not caches or registers

- **Computations**: flops
  - Only “useful” computations (usually floating-point) not address calculations, loop iterators..

**Arithmetic intensity**: flops / bytes
= computations / memory accesses
  - Property of the code

**Arithmetic throughput**: flops / s
  - Property of code + architecture
The roofline model

- How much performance can I get for a given arithmetic intensity?
  - Upper bound on arithmetic throughput, as a function of arithmetic intensity
  - Property of the architecture

Building the machine model

- Compute or measure:
  - Peak memory throughput
  - Ideal arithmetic intensity = peak compute throughput / mem throughput

  \[
  \text{GTX 980: } \frac{4612 \text{ (Gflop/s)}}{224 \text{ (GB/s)}} = 20.6 \text{ flop/B} \\
  \times 4 \text{ (B/flop)} = 82 \text{ (dimensionless)}
  \]

- Achievable peaks may be lower than theoretical peaks
  - Lower curves when adding realistic constraints

Beware of units:
float=4B, double=8B!
Using the model

- Compute arithmetic intensity, measure performance of program
- Identify bottleneck: memory or computation
- Take optimization decision

![Arithmetic throughput diagram]

- Arithmetic throughput (Gflop/s)
- Measured performance
- Optimize memory accesses
- Optimize computation
- Reuse data

Arithmetic intensity (flop/byte)
Example: dot product

```
for i = 1 to n
    r += a[i] * b[i]
```

- How many computations?
- How many memory accesses?
- Arithmetic intensity?
- Compute-bound or memory-bound?
- How many Gflop/s on a GTX 980 GPU?
  - With data in GPU memory?
  - With data in CPU memory?
- How many Gflop/s on an i7 4790 CPU?

GTX 980: 4612 Gflop/s, 224 GB/s
i7 4790: 460 Gflop/s, 25.6 GB/s
PCIe link: 16 GB/s
Example: dot product

```plaintext
for i = 1 to n
    r += a[i] * b[i]
```

- How many computations?  → 2 n flops
- How many memory accesses? → 2 n words
- Arithmetic intensity?   → 1 flop/word = 0.25 flop/B
- Compute-bound or memory-bound? → Highly memory-bound

How many Gflop/s on a GTX 980 GPU?
- With data in GPU memory? 224 GB/s × 0.25 flop/B → 56 Gflop/s
- With data in CPU memory? 16 GB/s × 0.25 flop/B → 4 Gflop/s

How many Gflop/s on an i7 4790 CPU?
25.6 GB/s × 0.25 flop/B → 6.4 Gflop/s

Conclusion: don't bother porting to GPU!

GTX 980: 4612 Gflop/s, 224 GB/s
i7 4790: 460 Gflop/s, 25.6 GB/s
PCIe link: 16 GB/s
Takeaway

- Result of many tradeoffs
  - Between locality and parallelism
  - Between core complexity and interconnect complexity
- GPU optimized for throughput
  - Exploits primarily DLP, TLP
  - Energy-efficient on parallel applications with regular behavior
- CPU optimized for latency
  - Exploits primarily ILP
  - Can use TLP and DLP when available
- Performance models
  - Back-of-the-envelope calculations and common sense can save time
- Next time: GPU programming in CUDA