Master Degree Program in Computer Science and Networking

High Performance Computing

Marco Vanneschi

MIMD, SIMD, GPU, and others

Technology space for parallel processing: status and trends

www.di.unipi.it/~vannesch  section: Teaching
Subject and goals of this seminar

• Technologies for parallel processing, with emphasis on single-chip architectures
  – SIMD, SIMT (GPU): introduction, main features, performance, and utilization
  – homogeneous vs heterogeneous MIMD
  – heterogeneous MIMD + SIMD, MIMD + SIMT

related to classes of applications.

• Integrated technology: status and trends of processor (core) and memory technology
  – the central problem is energy efficiency: area and power consumption (wrt performance)
  – number of cores per chip (core density)
  – impact of ILP, vectorization, multithreading, …, memory hierarchy
  – (interconnection technologies have already been studied).

• Programmability issues.

Where we are and where we are going
Comment of research ideas and market products

• Analysis based on facts and rationales.

• Some advanced R&D products will be mentioned that have not (not yet) been transferred into market products
  – for reasons independent of the technological relevance,
  – however, they can have a fundamental impact on future products.
  – Two notable historical examples:
    • Data-flow
    • VLIW

No pure data-flow or VLIW machine has been commercialy successful, however they have had a decisive impact on the most advanced commercial products (ILP, MIMD, SIMD, compilers, run-time supports, …).
### Classes of computations

- **Application graphs vs accelerators of sequential programs**
- **Stream vs single value**
- **Irregularity vs regularity**: in control flow and/or data

<table>
<thead>
<tr>
<th>Search engines</th>
<th>Graphics pipeline and image processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter analytics</td>
<td>Signal processing</td>
</tr>
<tr>
<td>Network monitoring and analysis</td>
<td>Physical simulations</td>
</tr>
<tr>
<td>Data- and web-mining</td>
<td>Computational biology</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Fluidodynamics</td>
</tr>
<tr>
<td>Compression</td>
<td>Some numerical analysis kernels (e.g. Cholewski)</td>
</tr>
<tr>
<td>Computational financing</td>
<td>Some regular numerical kernels</td>
</tr>
<tr>
<td>Data Stream Processing</td>
<td>Simple map-reduce analytics</td>
</tr>
</tbody>
</table>
### Homogeneous vs Heterogeneous Architectures

#### Homogeneous

**MIMD architecture with ‘general’ processors**
- uniform cooperation and programming mechanisms
- trade-off generality and programmability *vs* performance

- **Importance of ILP features of ‘general’ processors:**
  - *in each core*
    - Functional Units - EU
    - **Vector processing**
      - SSE, AVX (Intel), VIS (Sparc), …
      - *vs VPU* (IBM Power PC, Intel Larrabee, …)
    - Hardware multithreading
  - **General** memory hierarchy organization
    - Pros and Cons
  - **Potentially**, uniform programming frameworks and tools

#### Heterogeneous

**MIMD + distinct co-processors (accelerators)**
- reduce complexity of performance optimization and energy efficiency for specific (parts of) applications
- at the expense of generality (and programmability)

1. **Co-processors as I/O units or specialized processors in the same chip**
   - hopefully, uniform mechanisms (e.g., *IBM Wire-Speed*)

2. **SIMD (Single Instruction Stream Multiple Data Stream) machines**
   - *Data-parallel* computations delegated to (fine-grain) parallel co-processors
   - Old-style array processors (Connection Machine, Ape)
   - **GPUs** and extensions of SIMD into **SIMT** (Single Instruction Stream Multiple Threads)
Heterogeneous MIMD
Network processors

• Parallel on-chip co-processors oriented to network processing:
  – Real-time processing of multiple data streams
  – IP protocol packet switching and forwarding capabilities
    • Packet operations, queueing, Pattern matching, checksum / CRC
  – Tree searches
  – Frame forwarding
  – Frame filtering
  – Frame alteration
  – Traffic control and statistics
  – QoS control
  – Enhance security
  – Primitive network interfaces on-chip

• Intel, IBM, Ezchip, Xelerated, Agere, Alchemy, AMCC, Cisco, Cognigine, Motorola, ...
IBM Wire-Speed Processor (WSP)

• Heterogenous architecture

• 16 general-purpose multithreaded cores:
  – PowerPC, 2.3 GHz
  – In-order
  – SMT, 4 simultaneous threads/core

• Domain-specific co-processors (accelerators)
  – targeted toward networking applications:
    • packet processing, security, pattern matching, compression, XML
    • custom hardware-firmware components for optimizations
  – networking interconnect: four 10-Gb/s links
Figure 1

WSP functional diagram and characteristics. The A2 cores are described later in the text. (XML: Extensible Markup Language; Regex: regular expression and pattern-matching accelerator; Comp: compression and decompression accelerator; Crypto: cryptographic accelerator; MAC: media access control, a unique network adapter ID; MemCtrl: memory controller; QoS: quality of service; gen2: generation 2.)
<table>
<thead>
<tr>
<th>Accelerator unit</th>
<th>Algorithm</th>
<th>No. of engines</th>
<th>Projected bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Typical</td>
</tr>
<tr>
<td>HFEA</td>
<td>Network mode mode</td>
<td>4</td>
<td>40 Gb/s</td>
</tr>
<tr>
<td></td>
<td>Endpoint mode</td>
<td>4</td>
<td>40 Gb/s</td>
</tr>
<tr>
<td>Compression</td>
<td>Gzip (input bandwidth)</td>
<td>1</td>
<td>8 Gb/s</td>
</tr>
<tr>
<td></td>
<td>Gunzip (output bandwidth)</td>
<td>1</td>
<td>8 Gb/s</td>
</tr>
<tr>
<td>Encryption</td>
<td>AES</td>
<td>3</td>
<td>41 Gb/s</td>
</tr>
<tr>
<td></td>
<td>TDES</td>
<td>8</td>
<td>19 Gb/s</td>
</tr>
<tr>
<td></td>
<td>ARC4</td>
<td>1</td>
<td>5.1 Gb/s</td>
</tr>
<tr>
<td></td>
<td>Kasumi</td>
<td>1</td>
<td>5.9 Gb/s</td>
</tr>
<tr>
<td></td>
<td>SHA</td>
<td>6</td>
<td>23–37 Gb/s</td>
</tr>
<tr>
<td></td>
<td>MD5</td>
<td>6</td>
<td>31 Gb/s</td>
</tr>
<tr>
<td></td>
<td>AES/SHA</td>
<td>3</td>
<td>19–31 Gb/s</td>
</tr>
<tr>
<td></td>
<td>RSA and ECC (rates shown are for RSA 1,024-bit and RSA 2,048-bit keys)</td>
<td>3</td>
<td>45,000 and 7,260 Gb/s</td>
</tr>
<tr>
<td>XML</td>
<td>Customer workload</td>
<td>4</td>
<td>10 Gb/s</td>
</tr>
<tr>
<td></td>
<td>Benchmark</td>
<td>4</td>
<td>20 Gb/s</td>
</tr>
<tr>
<td>RegX</td>
<td>For typical pattern sets</td>
<td>8</td>
<td>20–40 Gb/s</td>
</tr>
</tbody>
</table>

Advanced features for *programmability* + *portability* + *performance*:

- **Uniform addressability: uniform virtual address space**
  - Every CPU core, accelerator and I/O unit has a separate MMU
  - Shared memory: NUMA architecture, including accelerators and I/O units (heterogeneous NUMA)
  - Coherent (snooping) and noncoherent caching support, also for accelerators and I/O
  - Result: accelerators and I/O are not special entities to be controlled through specialized mechanisms, instead they exploit the same mechanisms of CPU cores
    - *full process-virtualization of co-processors and I/O*

- **Special instructions for locking and core-coprocessor synchronization**
  - Load and Reserve, Store Conditional
  - Initiate Coprocessor

- **Special instructions for thread synchronization**
  - wait, resume
SIMD
SIMD architecture

**SIMD** (Single Instruction Stream Multiple Data Stream)

**Data-parallel (DP) paradigm at the firmware-assembler level**

- **SIMD instructions** (map, stencil, reduce)
- **Instruction & Data Memory**
- **Instruction Unit**
- **Execution Unit**
  - Local Memory
  - ... (multiple Execution Units)
  - Interconnection structure
- **Instruction issue**: broadcast/multicast
- Data distribution (scatter, multicast) and collection (gather)
- **DP processor cores (workers)**
  - e.g. 2-3 dimension mesh

- **Example**: IU controls the partitioning of a float vector into the local memories (scatter), and issues a **sequence of requests** of float operations (LDF/STF, ADDF, MULF, ...) to all EUs
- Pipelined processing IU-{EU}, pipelined EUs
- Extension: partitioning of IU-{EU} into disjoint subsets for DP multiprocessing (**MIMD + SIMD**)
SIMD: parallel, high-performance co-processor

- SIMD cannot be general-purpose: at most can be programmable.
- I/O bandwidth and latency for data transfer between Host and SIMD co-processor is critical.
- Challenge: proper utilization of central processors and peripheral SIMD co-processors for designing high-performance parallel programs.
Vector processing in non-SIMD machines

• **Vector instructions** of ‘general’ processor [Daniele’s seminar] are sometimes called *SIMD instructions*,
  – however they are *not* instructions of a SIMD machine.

• In both cases *data-parallelism* is exploited at the firmware level,
  – however *vector* instructions of a ‘general’ processor are *Instruction Level Parallelism* features: executed in a SSE/AVX/VIS fashion or in a **VPU** (double precision).
  – This is not an ‘accelerator’ approach.

• **VPU**: # vector operations per clock cycle ~ 20 times than SSE

• Present in almost all the MIMD multicore products
  – Intel Larrabee, Sandy Bridge, …
  – …
  – IBM Power, Cell, Wire-Speed, …

  *not* in Tilera.
Basic SIMD programming model

• A particular way of thinking about *data-parallel* programs executed through *sequences of SIMD instructions*.

• Instructions are sequentially ordered (Single Instruction Stream).

• Each instruction refers to distinct elements of one or more array/matrix data structure (Multiple Data Stream).

• In other terms, *at each step of the sequential program, all the data-parallel ‘workers’ (SIMD machine EUs) execute the same instruction on their own local data.*
**Example 1**

```plaintext
float A[M], B[M], C[M];

\forall i = 0 .. M - 1: C[i] = \sqrt{A[i] + B[i]}

Sequential compilation (VP[i]):

```
LOOP:
  LOADF A[i]
  LOADF B[i]
  ADDF A[i], B[i], c
  SQRT C[i], C[i]
  STOREF C[i]
  INCR i
  IF (i < M) LOOP
```

In a (MIMD) parallel program, we express this data-parallel map, with:
- [Scatter, Gather,]
- \(n\) sequential workers,
- \(A, B, C\) partitioned, with \(g = \frac{M}{n}\) float elements for each worker.
- Partitions may consist in consecutive elements, or interleaved/strided elements.
- Each worker executes the sequential program on the assigned partition.

In a SIMD machine, once data have been (are) partitioned in the local memories, the \(n\) EUs execute the computation in a step-by-step fashion, similarly to a long instruction word machine:

Step 1: LOADF ..., LOADF ..., ..., LOADF ...
Step 2: LOADF ..., LOADF ..., ..., LOADF ...
...
Let us refer to a loop-unrolled \( n \)-way parallel execution (partitioning by blocks):

| \( \text{LOADF } A[0] \) | \( \text{LOADF } A[g] \) | \( \ldots \) | \( \text{LOADF } A[(n-1)g] \) |
| \( \text{LOADF } B[0] \) | \( \text{LOADF } B[g] \) | | \( \text{LOADF } B[(n-1)g] \) |
| \( \text{ADDF } A[0], B[0], C[0] \) | \( \text{ADDF } A[g], B[g], C[g] \) | | \( \text{ADDF } A[(n-1)g], B[(n-1)g], C[(n-1)g] \) |
| \( \text{SQRT } C[0] \) | \( \text{SQRT } C[g] \) | | \( \text{SQRT } C[(n-1)g] \) |
| \( \text{STOREF } C[0] \) | \( \text{STOREF } C[g] \) | | \( \text{STOREF } C[(n-1)g] \) |
| \( \text{LOADF } A[1] \) | \( \text{LOADF } A[g+1] \) | | \( \text{LOADF } A[(n-1)g+1] \) |
| \( \text{LOADF } B[1] \) | \( \text{LOADF } B[g+1] \) | | \( \text{LOADF } B[(n-1)g+1] \) |
| \( \text{ADDF } A[1], B[1], C[1] \) | \( \text{ADDF } A[g+1], B[g+1], C[g+1] \) | | \( \text{ADDF } A[(n-1)g+1], B[(n-1)g+1], C[(n-1)g+1] \) |
| \( \text{SQRT } C[1] \) | \( \text{SQRT } C[g+1] \) | | \( \text{SQRT } C[(n-1)g+1] \) |
| \( \text{STOREF } C[1] \) | \( \text{STOREF } C[g+1] \) | | \( \text{STOREF } C[(n-1)g+1] \) |
| \( \ldots \) | \( \ldots \) | | \( \ldots \) |

The step-by-step execution of \textbf{SIMD instructions} is shown - issued by IU, executed by EUs - possibly pipelined with service time equal to one time slot.

The loop control ‘increment and if’ conditional instructions are executed \textbf{by IU in a centralized manner}: this is possible because it is a \texttt{for} loop and the \textit{branch condition} is common to all parallel executions (\textit{global predicate}) – once the loop terminates, no instruction is issued from IU to EUs.

\textbf{A n-way replicated (parallel) execution} of the loop termination condition is possible too, just because the conditions above (\texttt{for} loop, global predicate) can be consistently verified by all EUs.
Basic SIMD programming model

• The main reason for building a separate machine (instead of remaining inside the EU of an ILP CPU, i.e. VPU) is to take advantage of the potentially high parallelism of physically distinct ‘processors’ on regular data-parallel computations.

• Distinct ‘processors’ are EUs (with local registers), working step-by-step under the supervision of a centralized IU (instruction scheduling).
  – It is the same principle of superscalar, but now EUs work always in parallel and on the same instruction.

• Powerful floating point FUs (double precision)
  – including SQRT, SIN, COSIN, …, and so on.

• LOAD/STORE instructions are executed in parallel by all EUs between their local memory and their local registers.

• Data are transferred between the shared main memory (of the SIMD machine) and the local memories by proper strategies
  – on-demand, prefetching, caching if primitive at the firmware level - according to the regular memory model of the computation.
SIMD: why the expectation for very high parallelism

- The basic parallelism form is **Map**
  - *for* loops on regular data structures
  - possibly *reduce* may be present

- **Single-value** or, at most, stream-equivalent computations (e.g. stream of pixel rows of the same image), according to the ‘single-program accelerator’ approach

- As known, data-parallel *map* on large *single-value* data may have a *very high optimal parallelism degree*, even for fine grain calculations,
  - even *any* parallelism degree is welcome if the occurrence of data distribution/collection is a (very) rare events wrt to calculation

- In such cases, a high scalability is naturally expected
  - \( \sim 10^3 \) cores

- Potentials for good *energy efficiency* (simple cores: EUs).
Example 2: the divergent branch problem

float A[M], B[M], C[M];
∀ i = 0 .. M – 1: C[i] = sqrt (max (A[i], B[i]))

Sequential compilation (VP[i]):

LOOP:
  LOADF A[i]
  LOADF B[i]
  **IFF (A[i] ≥ B[i]) THEN**
    SQRT B[i], C[i]
    STOREF C[i]
  GOTO CONT

THEN:
  SQRT A[i], C[i]
  STOREF C[i]

CONT:
  INCR i
  IF (i < M) LOOP

The **IFF** instruction is executed by IU in a centralized manner. For each iteration, **only a fraction of EUs is active** in executing the then-else alternative code branches. Thus parallelism explicitation is lower in alternative code branches.

Moreover, the IU is also in charge of controlling the termination of the selected branch.

If the conditional instruction **IFF** is executed in parallel by all the EUs, without centralized IU assistance, some of them are able to continue, while the others are blocked: they must wait until the selected branch is terminated. Thus, a non-trivial **global synchronization** is needed. This is the **GPU** case: each instruction ‘crosses’ several threads (SIMT).
The divergent branch problem

• Parallelism in presence of conditional branches is a classical problem in concurrent semantics.
  – An imperative language forces the introduction of a logically centralized control.
  – More in general, the same principle applies to the nondeterminism semantics (e.g., CSP/LC alternative command implementation).
  – By definition, a functional language is not affected by this problem, even at the instruction level, e.g. data-flow model.

• Any (imperative) solution lowers the scalability of architectures (like GPU) trying to exploit very high parallelism through the SIMD approach.
  – A lot of research work has been done on this subject (e.g. dynamic instruction formation, two-level scheduling, and so on), however solutions lead to non-trivial complications of the architecture and programming model.

• This problem does not exist in MIMD parallelism: each worker is a process/thread with its own ‘locus of control’ – each instruction belongs to one and only one thread.
  – Thus, the divergent branch problem does not exist in non-SIMD Vector Processing (by definition, conditional instructions are executed by IU without any visibility by EU and VPU).
**SIMD parallelism and scalability**

- SIMD array processors are not necessarily limited to *map* computations.

- **Stencils** may be target computations, as well as complex *reduce* and *parallel prefix*.
  - This was the computational target (and the *fundamental scientific contribution*) of the MIT **Connection Machine** (up to 65K one-bit processors).
  - Other SIMD machines were oriented to map and **static fixed stencils**, e.g., Ape executing a 4-neighbours convolution algorithm (Jacobi-like) for High Energy Physics simulations.

- For **GPUs**, which originate from the special-purpose graphics co-processors, algorithms are basically *map*.
  - Moreover, the **CUDA** programming model (NVIDIA) rely heavily on the underlying architecture with *shared memory*, so many stencils (not all !) can be modeled as *map + barrier*. 
SIMD: ‘extensions for generality’, programmability

- Of course, many other computations have the same or similar structure wrt the original target of special-purpose machines,
  - thus SIMD can have a wider application area than the original one,
  - so some forms of **programmability** must be provided to the user.
  - However, absolutely **this does not mean that SIMD are general-purpose machines**.

- What about **true** stream-based computations?
- What about **irregular and/or dynamic** computations (control structure and/or data structure)?
- What about **compositionality**?
- What about **portability**?
GPU
From special-purpose GPU to GP-GPU

First version of NVIDIA Tesla.

Special purpose, on-chip GRAPHICS PIPELINE.

Each pipeline stage is a data-parallel module, that can be implemented by a SIMD machine.

Simple form of MIMD+SIMD.

Load balancing between pipeline stages (parallelism degree of the single data-parallel modules) can be achieved for known workloads (e.g. more pixels than vertices).

[M. Arora, 2013]
Programmable ‘GP’-GPU

- New NVIDIA Testa and AMD solutions.
- Architectural supports, and multithreading, to achieve a certain degree of programmability.
- 2-D and 3-D graphics.
  - Other applications have characteristics similar to graphics computations.
- **Basically: map operating on multiple-dimension arrays.**
  - Conditional control flow issues: see the divergent problem of the SIMD model.
- **NVIDIA CUDA, AMD CTM programming frameworks.**
  - In the simplest utilization mode, a CUDA programmer writes the generic thread of the data parallel program (~ ‘virtual processor’), which is instanced by many threads on the actual machine.
  - *Stencils* have not a clear semantics, sometimes are difficult to be expressed in this programming model, and anyway are left to the programmer responsibility.
  - *Explicit parallelism and synchronization APIs* are provided:
    - barriers, atomic memory operations, and others. Difficult to use.
  - *Explicit memory hierarchy management APIs* are provided.
    - Difficult to use.
GPU on-chip architecture (NVIDIA Tesla)

‘Streaming’ multiprocessor: generalization of SIMD architecture to include hardware multithreading

SIMT (Single Instruction Multiple Thread)

[M. Arora, 2013]
SM - SIMT architecture

Banked Register File
(Fermi: 21 registers per thread = 128K registers)

Warp Scheduler (IU)

Operand Buffering

32 SIMT ‘Lanes’ (EUs)

FUs

SFUs

MEM

TEX

Shared Memory / L1 Cache

Single- and double-precision multiply-add.
(Replaces VPU)

[M. Arora, 2013]
Let us organize our data-parallel computation in a (possibly large) number of *functionally identical* hardware *threads*, e.g.

\[ \forall \ i = 0 .. \ M - 1: \ C[i] = \sqrt{A[i] + B[i]} \]

A SM is composed of 32 independent cores (EUs, or ‘lanes’ in GPU terminology), each one mapping a hardware thread.

A **SIMT instruction** (generalization of SIMD instruction) is composed by 32 scalar instructions, one for each thread.

Thus, 32 instructions per time slot are issued.

A group of 32 threads is called ‘**Warp**’.

Fermi has up to 48 Warps per SM = 1536 threads per SM.
Instruction scheduling and multithreading

Instruction scheduling (i.e. IU issuing) happens at the *granularity of Warps*. *All threads in a Warp execute together* using a *common program counter*.

- **Instruction processing:** *in-order* within a Warp.
- *Warps can be selected out-of-order:*
  - **Memory access latency hiding**
    - When a Warp incurs a delay (external memory access), the SIMT instruction of another active Warp is selected,
    - i.e., since all Warps are identical, hardware multithreading is used for latency hiding in a natural manner.
- **A single program counter per Warp:**
  - consistent with the SIMD approach,
  - simplifies the multiple context implementation wrt ‘general’ multithreading.
NVIDIA GeForce 8800 Tesla

- 16 SMs
- 32 cores per SMs
- 512 cores per chip
- 6 external memory interfaces
- Shared L2 cache
The divergent branch problem

From SIMD slides:

- If the conditional instruction IFF is executed in parallel by all the EUs (Warp lanes), without centralized IU (Warp scheduler) assistance, some of them are able to continue, while the others are blocked: they must wait until the selected branch is terminated. Thus, a non-trivial global synchronization is needed. This is the GPU case: each instruction ‘crosses’ several threads (SIMT).

- A lot of research work has been done on this subject (e.g. dynamic instruction formation, two-level scheduling, and so on), however solutions lead to non-trivial complications of the architecture and programming model.

- This problem does not exist in MIMD parallelism
GPU architecture and energy efficiency

- Efficient memory hierarchy for the specific class of computations:
  - 6 high-bandwidth DRAMs, on-chip L2 cache.
  - Local memory per thread, global memory per Warp.
  - Multithreading-based latency hiding is defined according to the (sufficiently) predictable memory model.

- Thread switching: every 2 – 4 clock cycles.

- The simplicity of cores reduces power consumption and allows GPU to exploit a massively large number of cores, i.e. order $10^3$ (~ 500 in GeForce 8800 Tesla, ~ 1000 in GTX 280 Tesla and GF 100 Fermi, ~ 2900 in GK 110 Kepler).

- However, the architectural solutions for
  - dynamic SIMT instruction formation (not VLIW)
  - divergent branch problem
  - intra-SM synchronization; inter-SM synchronization is not primitive

re-introduce a certain level of complexity, though the whole architecture can be classified as energy efficient.
CPU-GPU integration (1)

CPU (multiprocessor) + GPU as external co-processor (2006-2010)

[M. Arora, 2013]
Overcoming the latency and bandwidth problems in ‘CPU-GPU’ interconnect.

- On the other hand: the I/O Bus obsolescence has been highlighted also by the cluster interconnection technologies (Infiniband).

Current generation: CPU + GPU as on-chip co-processor /accelerator

[M. Arora, 2013]
CPU-GPU integration (3)

• Single-chip integration: other products
  – AMD Fusion, Intel Sandy Bridge.

Heterogeneous architecture on-chip

• Some preliminary benchmarks are promising.
• Several R&D issues on how to distribute work between ‘CPU’ (MIMD) and GPU/SIMD
• CPU simply ‘helps’ GPU problems to be solved?
• CPU computation part is less parallel than GPU one?
• Programming model still unclear.
CPU-GPU integration (4)

Data-parallel program: Virtual Processors version

Heterogeneous architecture: mapping problem
• how to distribute work to heterogenous parts, statically or dinamically

CPU-GPU interconnect and interaction mechanisms

Shared memory synchronization: too simplistic panacea?
Highly parallel MIMD
Basic problem

• Is it feasible to build ‘general-purpose’ MIMD machines with a very high number of cores per chip?
  – comparable to SIMD/GPU

• Is it feasible to achieve good energy efficiency with $10^3$ ‘general’ cores per chip?

• Which kind of multiprocessor?
Energy efficient ‘general’ processors

- **In-order vs out-of-order**
  - *In-order* exploits compile-time reordering and FIFO out-of-ordering (no additional firmware complexity)
  - *Out-of-order* includes general data-flow execution and branch prediction (serious additional firmware complexity)

- *With the same area and power:*
  
  \[
  \# \text{ in-order cores} / \# \text{ out-of-order cores} = \text{from 5 to 10}
  \]

- References: Larrabee (x86), IBM Power, Cell, Wire-Speed
  - they are examples of in-order MIMD architectures – other examples exist

- **RISC vs CISC**: the ratio increases at least by a factor **2-3**.
  - unless the area gain is used in RISC for
    - additional FUs, VPUs (1/3 area growth for VPU),
    - register arrays, cache capacity,
    - and/or additional (hopefully ‘smart’) interconnection.
Energy efficient highly parallel MIMD

*In-order vs out-of-order*

**Hardware multithreading**

- Area and power are not substantially affected by hardware multithreading
  - < 10% area or power increase per thread (wrt 30 % average bandwidth improvement (?)
- With multithreading, *out-of-order* performance gain is negligible for single programs.

*Rationale on status and trends:*

**# in-order RISC multithreaded cores per chip: 10^3** is realistic with current ULSI technology:

- Better *throughput* than out-of-order CISC, with the same area and power
- *Raw computational latency* (Floating Point) is re-gained through additional FUs and VPPUs
- *Compiler* technology is strategic (again) - VLIW
- Not severely limited by the application class (trade-off generality vs performance)

- Current examples: Adapteva Epiphany, ST Microelectronics, and others.
Epiphany – 4K RISC cores

Clean slate RISC Processor Designed for manycore
Network-On-Chip Designed for Real Time Applications
The special glue that makes manycore work.

Coprocessor to ARM/Intel CPU
25mW per core
Ease To Use
Energy efficiency: the memory impact

• Lower power consumption of memory chips
  – Historically, this motivated the large adoption of private or shared L2 cache on chip
  – L2 area is at least 30% - 50%

• Local memory vs caching
  – Current commercial MIMD multicore have few external memory interfaces and only L1-L2-(L3) on-chip cache

• NUMA architecture to be re-evaluated for multicore
  – Owing to the better (and still partially unexplored) capabilities for high parallelism and reduced memory contention
  – Local memory managed by program
    • Caching and cache-coherence by program (non automatic)
    • Memory model for application classes (locality, reuse, prefetching, latency hiding, stream computations, parallelism degree impact on memory allocation, ...)
    • Compiler and programming model impact
    • Message-passing vs shared variable support to parallelism forms

• NUMA examples: Cell, Epiphany, ...

• Also: larger register arrays

• Moreover, SIMD and SIMT architectures themselves rely substantially on local memory + large register arrays
  – Simple, specific memory models for specific application classes, which work well with NUMA too.
IBM Cell
MIMD ‘general’ applications and programming model

- **Stream-based generic graphs**
  - composability application-wide
  - optimizations

- **Farms and farm-like computations**
  - not only regular arrays
  - load balancing

**Data parallelism**

- **Regular map**: for loops or while with global conditions (reduce)
  - suitable for SIMD/SIMT too,
  - however, $O(N)$ algorithms are critical for SIMD/SIMT.

- **Generic control structures** (if, case, while with local conditions, complex reduce cases, prefix)
  - not suitable for SIMD/SIMT (centralized or low parallelism execution + synchronization overhead).

- **Map with replication**

- **Stencils**

- **Complex combinations of map, stencil, reduce, multicast**
  - Loop parallelism and accelerators vs true parallel programs.

- **Stream processing**
  - not limited to the stream-equivalent case only (SIMD/SIMT).

- **Memory models associated to data-parallel forms**
  - NUMA and message-passing

- **No explicit synchronization** in non-trivial loops:
  - CUDA and other shared memory tools: difficult to use,
  - explicit synchronization is performance degradation prone:
  - not always multithreading can alleviate the software lockout - to be not confused with memory latency masking.

*M. Vanneschi - High Performance Computing course*
Conclusion

• Highly-parallel on-chip architectures for high performance and energy efficiency
  – two main approaches

• Homogeneous MIMD with VPUs - or possibly heterogeneous MIMD with ‘uniform’ co-processors (customer-defined)
  – Potentials for technological feasibility with in-order RISC multithreaded cores
  – Challenges in memory models and on-chip NUMA architecture
  – Application generality and uniform programming model
  – Compiler technology evolution is needed for energy efficiency.

• MIMD + SIMD (SIMT)
  – Technological feasibility already demonstrated
  – Generality vs performance trade-off: far from being formalized and solved
  – Much more programmability is needed
  – More advanced software technology and compiler evolution are needed.
The first message-passing processor for parallel processing ...