



NATIONAL RESEARCH  
UNIVERSITY



# Computer Architecture and Operating Systems

## Lecture 13: Data-level parallelism: Vector, SIMD, GPU

**Andrei Tatarnikov**

[atatarnikov@hse.ru](mailto:atatarnikov@hse.ru)

[@andrewt0301](https://twitter.com/andrewt0301)

# Data-Level Parallelism

- **Data-level parallelism** is parallelism achieved by performing the same operation on independent data
- Best in dealing with arrays in for loops and processing other kinds of **identically structured data**
- Unsuitable for **control flow structures**

# Instruction and Data Streams

- An alternate classification

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

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

# Types of Parallel Processing

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each has an associated data memory, so that instructions are executed on different sets of data by different processors. Vector and array processors fall into this category.
- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Not commercially implemented.
- **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category.

# 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 RISC-V
  - v0 to v31:  $32 \times 64$ -element registers, (64-bit elements)
  - Vector instructions
    - `fld.v`, `fsd.v`: load/store vector
    - `fadd.d.v`: add vectors of double
    - `fadd.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

# Example: DAXPY ( $Y = a \times X + Y$ )

## ■ Conventional RISC-V code:

```
      fld      f0,a(x3)      # load scalar a
      addi     x5,x19,512    # end of array x
loop: fld      f1,0(x19)     # load x[i]
      fmul.d   f1,f1,f0      # a * x[i]
      fld      f2,0(x20)     # load y[i]
      fadd.d   f2,f2,f1      # a * x[i] + y[i]
      fsd      f2,0(x20)     # store y[i]
      addi     x19,x19,8      # increment index to x
      addi     x20,x20,8      # increment index to y
      bltu     x19,x5,loop    # repeat if not done
```

## ■ Vector RISC-V code:

```
      fld      f0,a(x3)      # load scalar a
      fld.v     v0,0(x19)     # load vector x
      fmul.d.vs v0,v0,f0      # vector-scalar multiply
      fld.v     v1,0(x20)     # load vector y
      fadd.d.v  v1,v1,v0      # vector-vector add
      fsd.v     v1,0(x20)     # store vector y
```

# 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

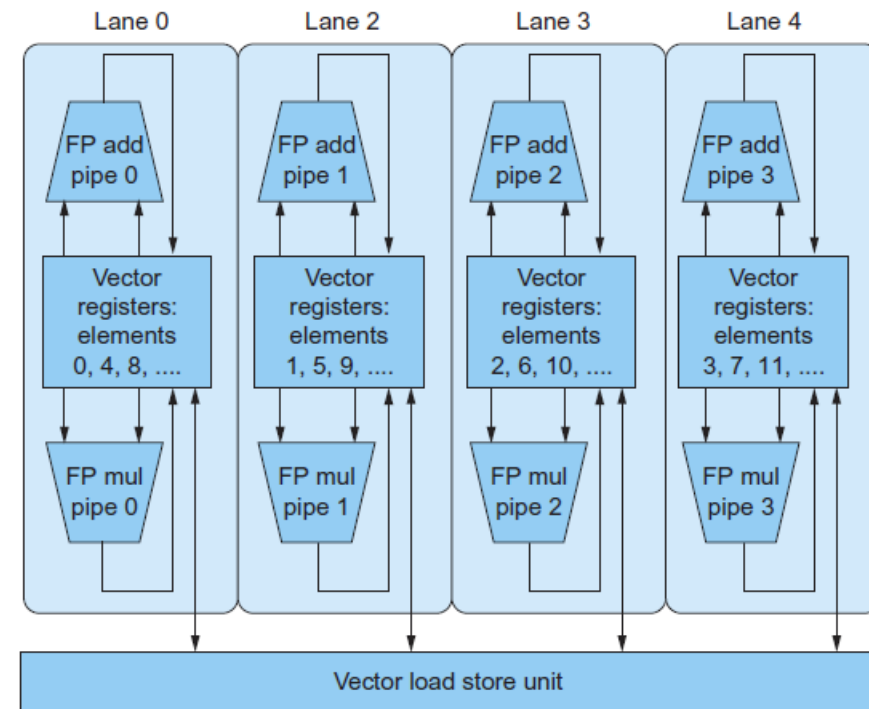
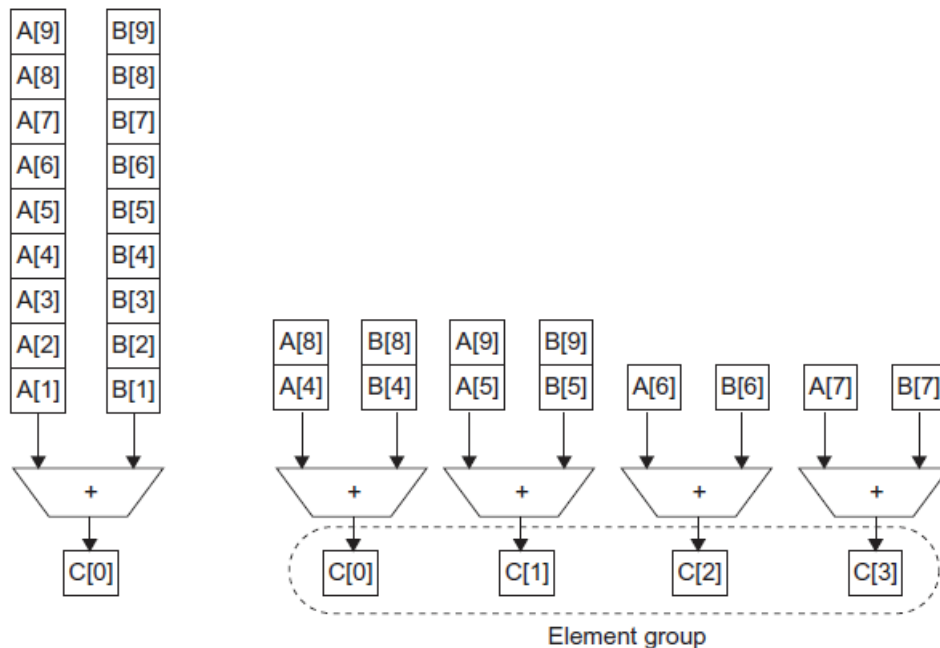
# SIMD

- 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 vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:



# GPU Architectures

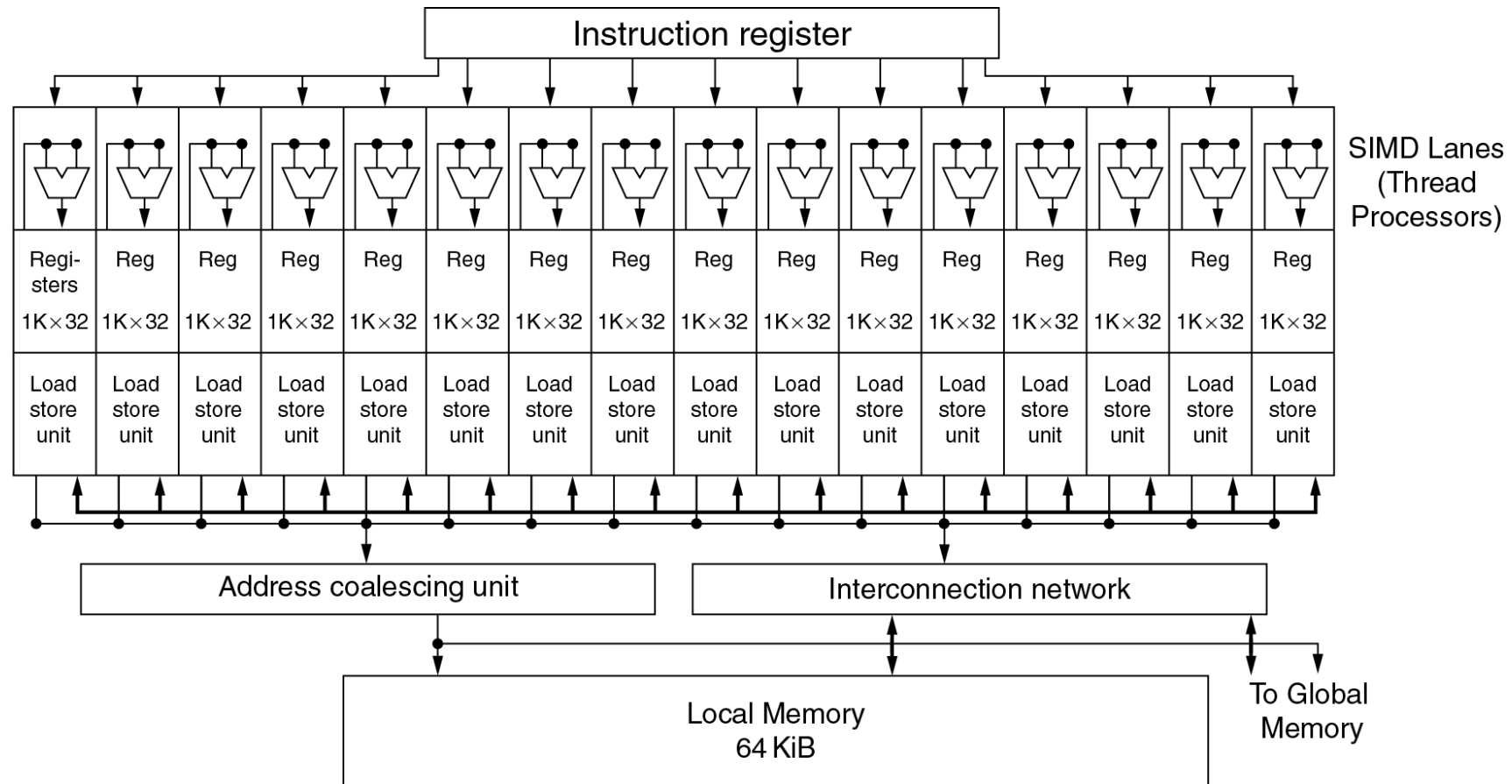
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

# History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law  $\Rightarrow$  lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

# Example: NVIDIA Fermi

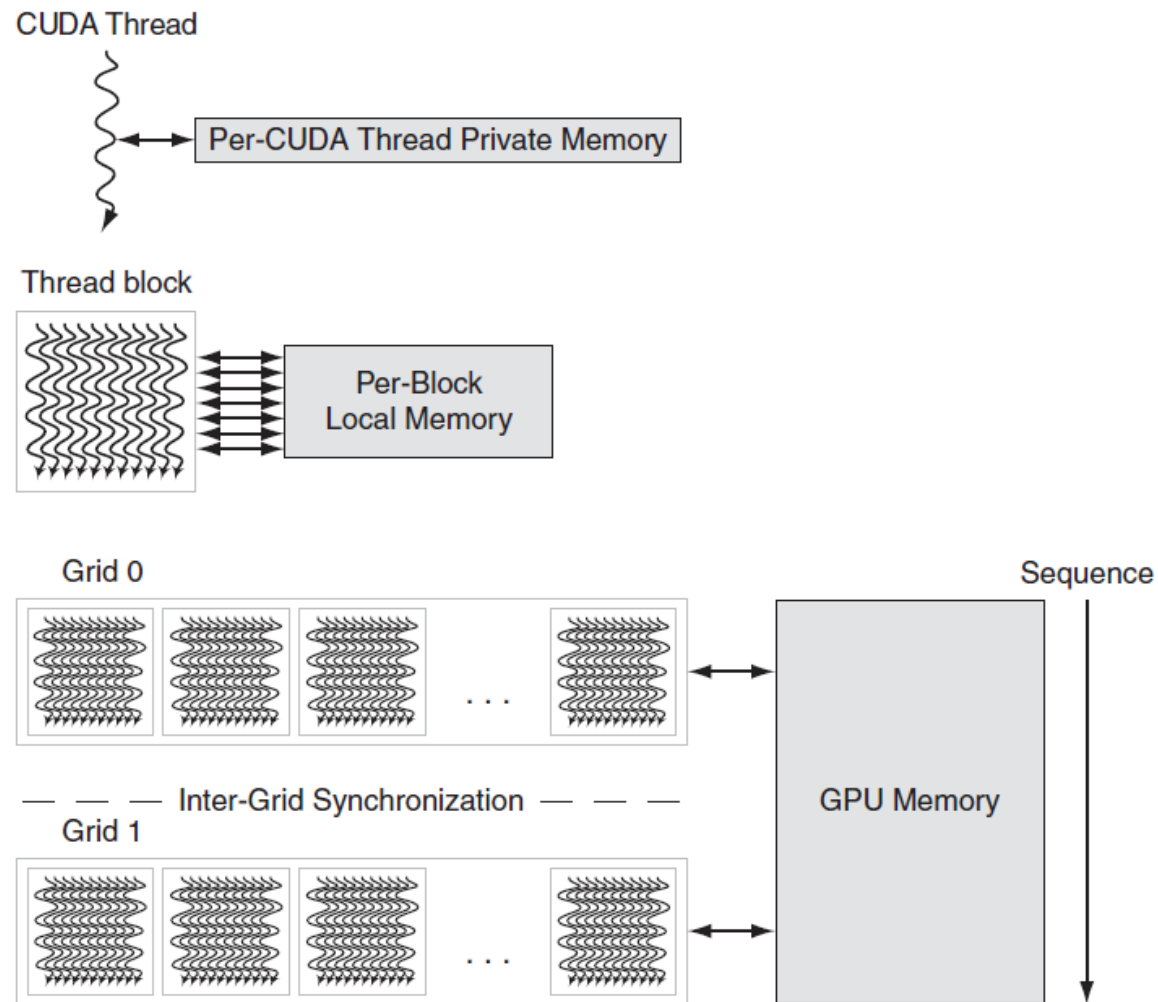
- Multiple SIMD processors, each as shown:



# Example: NVIDIA Fermi

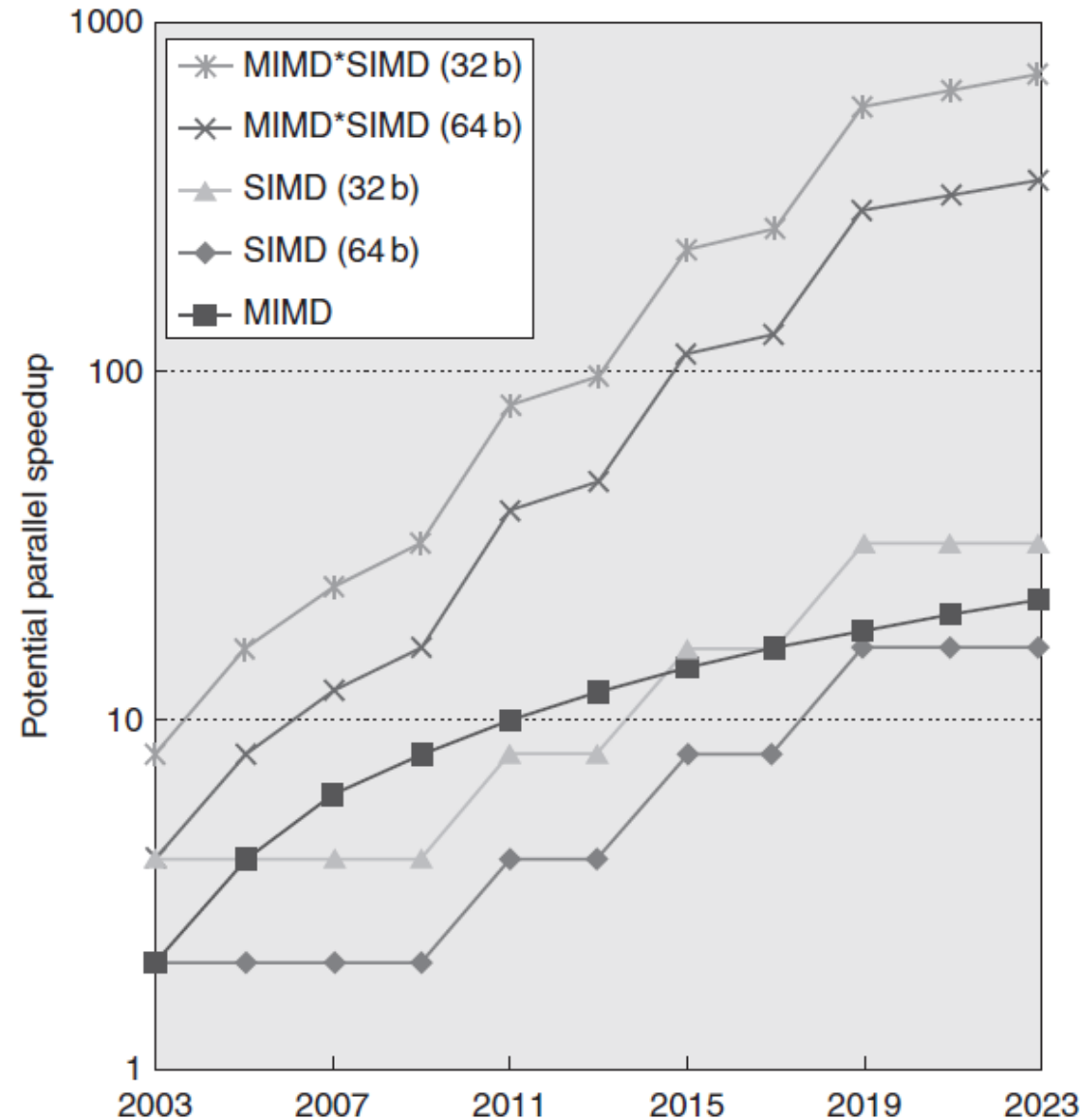
- SIMD Processor: 16 SIMD lanes
- SIMD instruction
  - Operates on 32 element wide threads
  - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
  - 64 registers per thread context

# GPU Memory Structures



# Concluding Remarks

- **SIMD** and **vector** operations match multimedia applications and are easy to program



# Any Questions?

```
                .text
__start:      addi t1, zero, 0x18
                addi t2, zero, 0x21
cycle:        beq t1, t2, done
                slt t0, t1, t2
                bne t0, zero, if_less
                nop
                sub t1, t1, t2
                j cycle
                nop
if_less:      sub t2, t2, t1
                j cycle
done:         add t3, t1, zero
```