Data-Level Parallelism

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

- Data-level parallelism (DLP)
- DLP architectures
- Vector architectures
- Vector architecture challenges
- Supercomputers
- SIMD extensions for multimedia
- Graphics processing units
- NVIDIA GPU systems

Types of Parallelism

- Instruction-level parallelism (**ILP**)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW).
- Data-level parallelism (**DLP**)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution).
- Thread-level parallelism (**TLP**)
  - Execute independent instruction streams in parallel (multithreading, multiple cores).
- Which is easiest to program?
- Which is most flexible form of parallelism?
  - I.e., can be used in more situations
- Which is most efficient?
  - I.e., greatest tasks/second/area, lowest energy/task

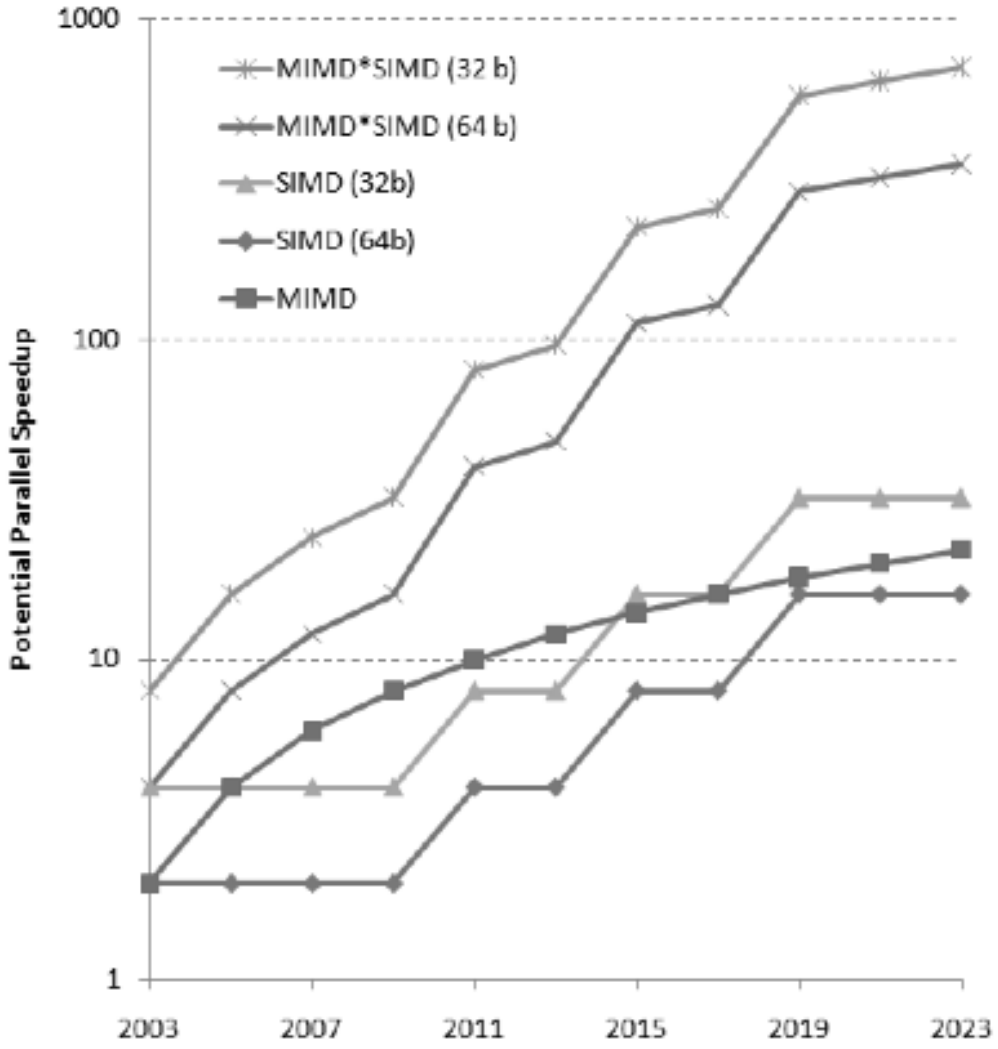
Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice.

- New applications, such as graphics, machine vision, speech recognition, and machine learning, all require large numerical computations that are often trivially data parallel.

- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are the most efficient way to execute these algorithms.

DLP Important for Conventional CPUs, Too



- Prediction for x86 processors.
- TLP: 2+ cores/two years.
- DLP: 2x width/four years.
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD ➔ (DLP)
  - MIMD ➔ (TLP)

ENGINEERING@SYRACUSE

DLP Architecture

Graphics Processing Units (GPUs)

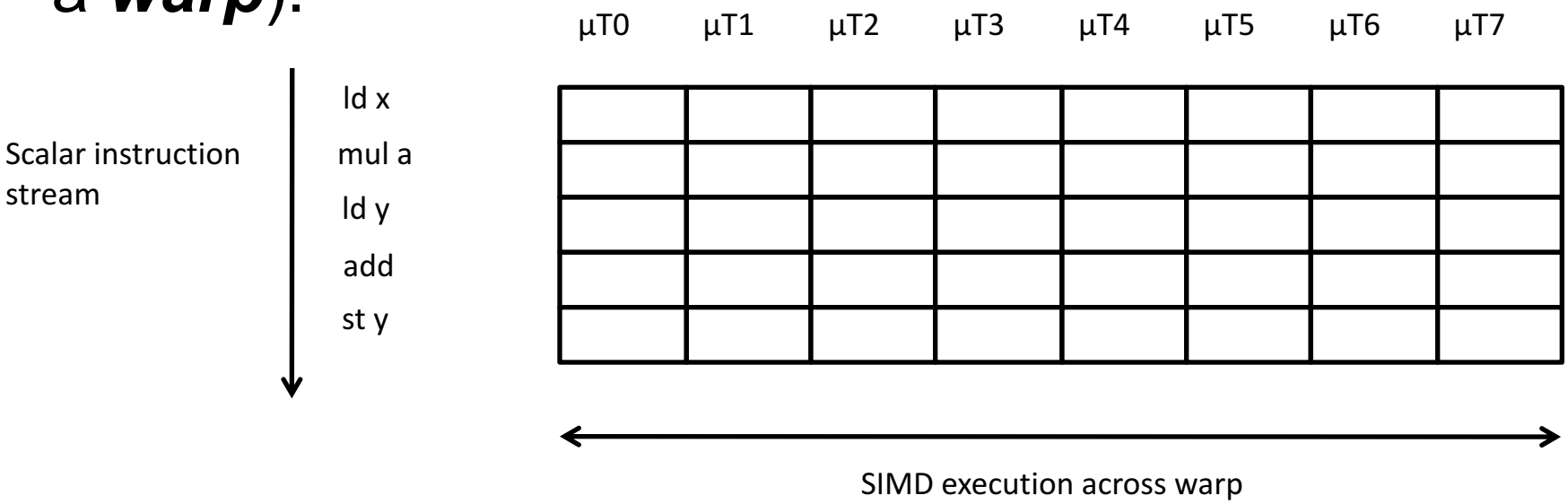
- Original GPUs were dedicated fixed-function devices for generating 3-D graphics (mid- to late 1990s) including high-performance floating-point units.
  - Provide workstation-like graphics for PCs.
  - User could configure graphics pipeline but not really program it.
- Over time, more programmability added (2001–2005).
  - E.g., new language CG for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images and computation to vertex and pixel shading computations.
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

General-Purpose GPUs

- In 2006, NVIDIA introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture."
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing.
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution.

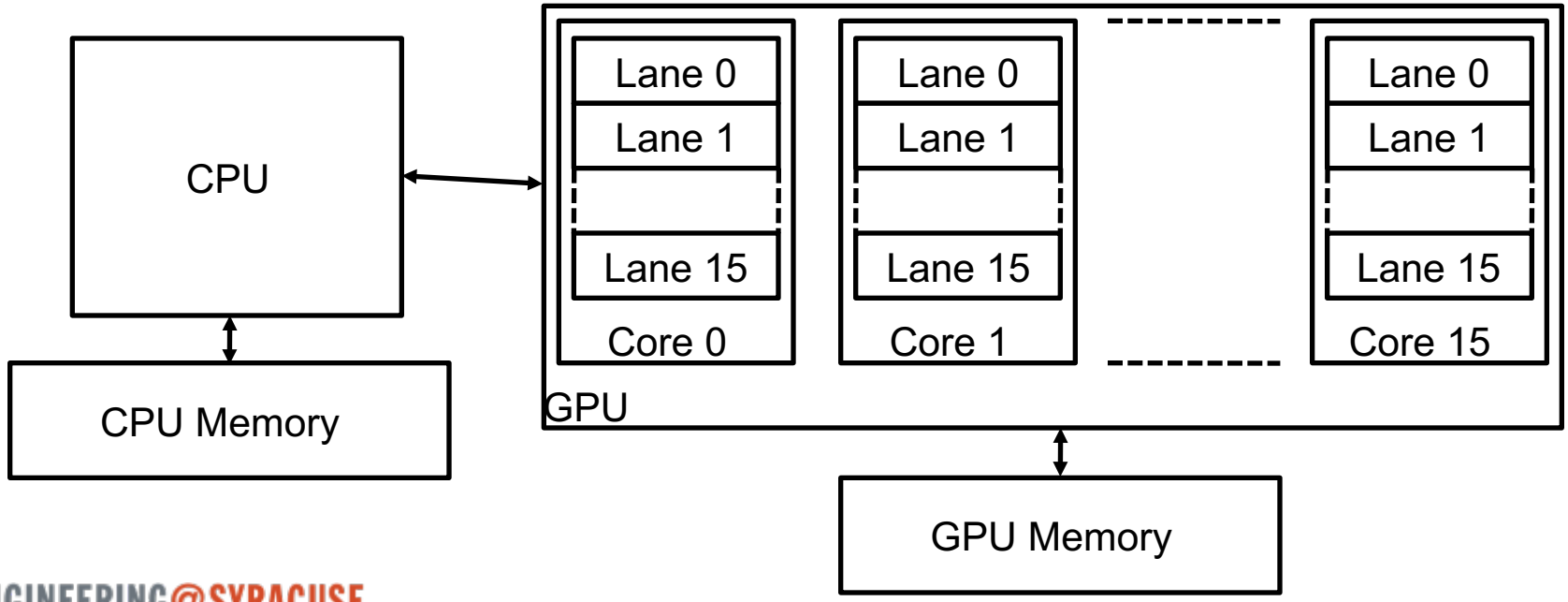
Single Instruction, Multiple Thread

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a ***warp***).



Hardware Execution Model

- GPU is built from multiple parallel cores, and each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor.
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core).
  - Programmer unaware of number of cores.



CUDA Programing Model

- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks.*

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++) y[i] = a*x[i] + y[i]; }
```

```
// CUDA version.
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y); // name, dim., parameters

__device__  // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
N thread
```

SIMD Architectures

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- SIMD is more energy efficient than MIMD.
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially.

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics processor units (GPUs)

- For x86 processors
  - Expect two additional cores per chip per year.
  - SIMD width to double every four years.
  - Potential speedup from SIMD to be twice that from MIMD!

ENGINEERING@SYRACUSE

Vector Architectures

Vector Architectures

- Basic idea:
  - Read sets of data elements into “vector registers.”
  - Operate on those registers.
  - Disperse the results back into memory.

- Registers are controlled by compiler,
  - Used to hide memory latency.
  - Leverage memory bandwidth.

VMIPS

- Example architecture: VMIPS
  - Loosely based on Cray-1
  - Vector registers
    - Each register holds a 64-element, 64 bits/element vector.
    - Register file has 16 read ports and eight write ports.
  - Vector functional units
    - Fully pipelined.
    - Data and control hazards are detected.
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers

VMIPS Instructions

- ADDVV.D: add two vectors.
- ADDVS.D: add vector to a scalar.
- LV/SV: vector load and vector store from address

- Example: DAXPY

```
L.D       F0,a       ; load scalar a
LV        V1,Rx      ; load vector X
MULVS.D   V2,V1,F0   ; vector-scalar multiply
LV        V3,Ry      ; load vector Y
ADDVV     V4,V2,V3   ; add
SV        Ry,V4      ; store the result
```

- Requires six instructions vs. almost 600 for MIPS

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle.
  - Execution time is approximately the vector length.
- *Convey:*
  - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convey via *chaining.*

- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

- *Chime*
  - Unit of time to execute one convey.
  - *m* conveys executes in *m* chimes.
  - For vector length of *n*, requires *m* x *n* clock cycles.

Example

```
LV          V1,Rx       ;load vector X
MULVS.D     V2,V1,F0    ;vector-scalar multiply
LV          V3,Ry       ;load vector Y
ADDVV.D     V4,V2,V3    ;add two vectors
SV          Ry,V4       ;store the sum
```

```
Convoys:
1     LV          MULVS.D
2     LV          ADDVV.D
3     SV
```

- Three chimes, two FP ops per result, cycles per FLOP = 1.5.
- For 64 element vectors, requires 64 x 3 = 192 clock cycles.

ENGINEERING@SYRACUSE

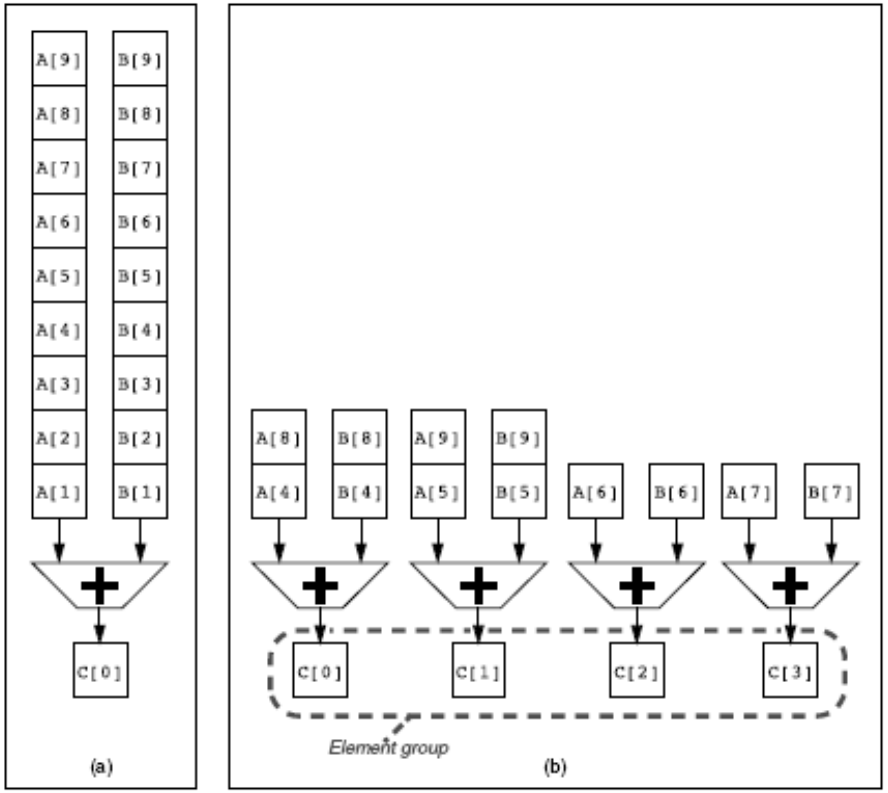
Vector Architecture Challenges

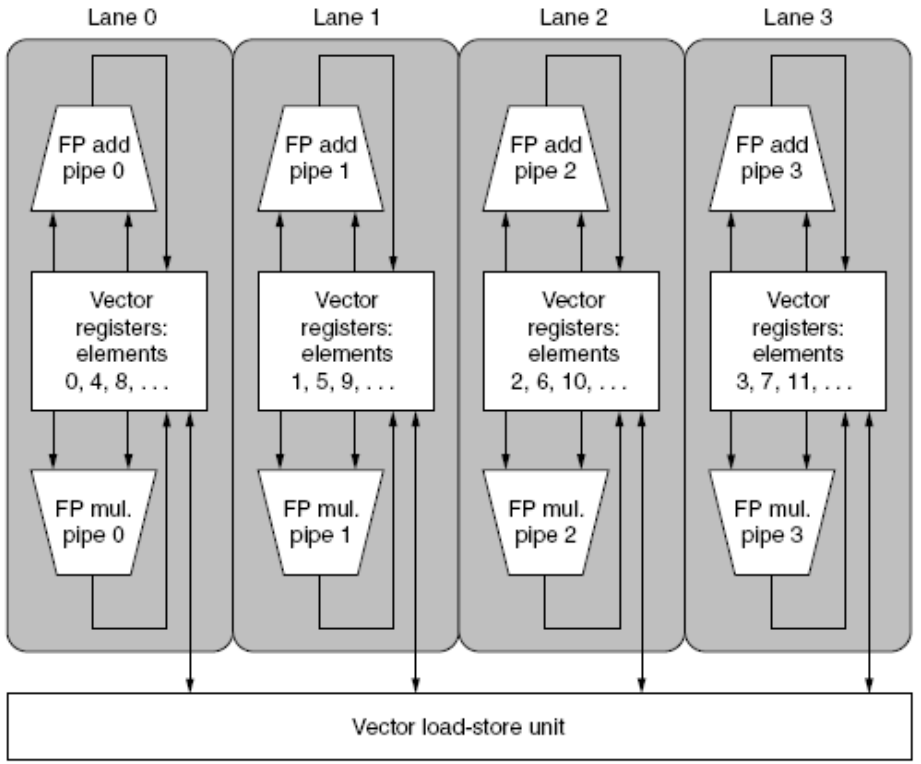
Challenges

- Start up time
  - Latency of vector functional unit.
  - Assume the same as Cray-1.
    - Floating-point add => six clock cycles
    - Floating-point multiply => seven clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements
  - > One element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

Multiple Lanes

- Element *n* of vector register *A* is “hardwired” to element *n* of vector register *B.*
  - Allows for multiple hardware lanes

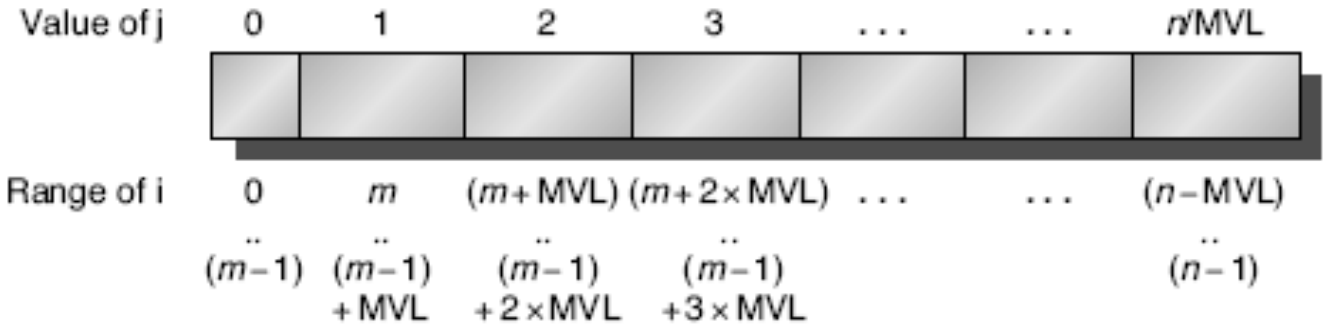




Vector Length Register

- Vector length not known at compile time?
- Use Vector length register (VLR)
- Use strip mining for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
        for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
                Y[i] = a * X[i] + Y[i] ; /*main operation*/
        low = low + VL; /*start of next vector*/
        VL = MVL; /*reset the length to maximum vector length*/ }
```



Vector Mask Registers

- Consider:

```
for (i = 0; i < 64; i=i+1)
      if (X[i] != 0)
            X[i] = X[i] – Y[i];
```

- Use vector mask register to “disable” elements:

```
LV          V1,Rx        ;load vector X into V1
LV          V2,Ry        ;load vector Y
L.D         F0,#0        ;load FP zero into F0
SNEVS.D     V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2     ;subtract under vector mask
SV          Rx,V1        ;store the result in X
```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores.
- Spread accesses across multiple banks.
  - Control bank addresses independently.
  - Load or store nonsequential words.
  - Support multiple vector processors sharing the same memory.

- Example:
  - 32 processors, each generating four loads and two stores/cycle.
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns.
  - How many memory banks needed?

Stride

- Consider:

```
for (i = 0; i < 100; i=i+1)
   for (j = 0; j < 100; j=j+1) {
       A[i][j] = 0.0;
       for (k = 0; k < 100; k=k+1)
          A[i][j] = A[i][j] + B[i][k] * D[k][j]; }
```

- Must vectorize multiplication of rows of B with columns of D.
- Use *nonunit stride.*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM (stride, #banks) < bank busy time
  - For any stride s and number of banks M, a stall will occur

    if LCM (s, M) / s ≤ mem-access latency.

Scatter-Gather

Want to vectorize loops with indirect accesses:

```
for (i = 0; i < n; i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

Indexed load instruction (gather)

```
LV          Vk, Rk          ;load K
LVI         Va, (Ra+Vk)     ;load A[K[]]
LV          Vm, Rm          ;load M
LVI         Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D     Va, Va, Vc      ;add them
SVI         (Ra+Vk), Va     ;store A[K[]]
```

Programming Vector Architecture

- Compilers can provide feedback to programmers.
- Programmers can provide hints to compiler.

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

ENGINEERING@SYRACUSE

Supercomputers

Supercomputers

- Definition of a supercomputer:
  - Fastest machine in the world at a given task
  - A device to turn a compute-bound problem into an I/O bound problem
  - Any machine costing $30M+
  - Any machine designed by Seymour Cray
- CDC6600 (Cray 1964) regarded as first supercomputer.

Supercomputer Applications

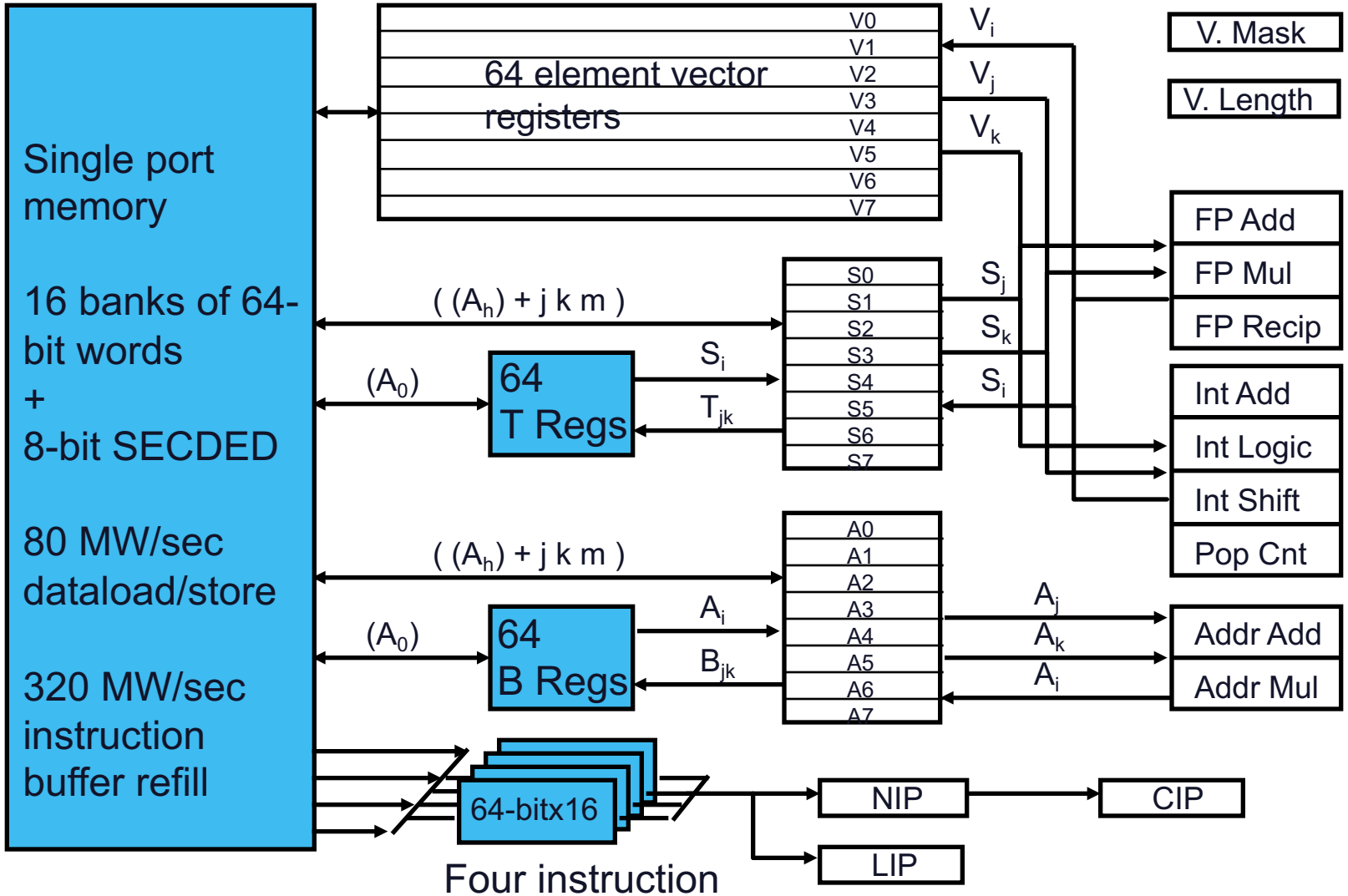
- Typical application areas:
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car-crash simulation)
- All involve huge computations on large datasets.
- In 1970s and 1980s, supercomputer = vector machine.

Vector Supercomputers

- Epitomized by Cray-1, 1976:
- Scalar unit + vector extensions
  - Load/store architecture
  - Vector registers
  - Vector instructions
  - Hardwired control
  - Highly pipelined functional units
  - Interleaved memory system
  - No data caches
  - No virtual memory

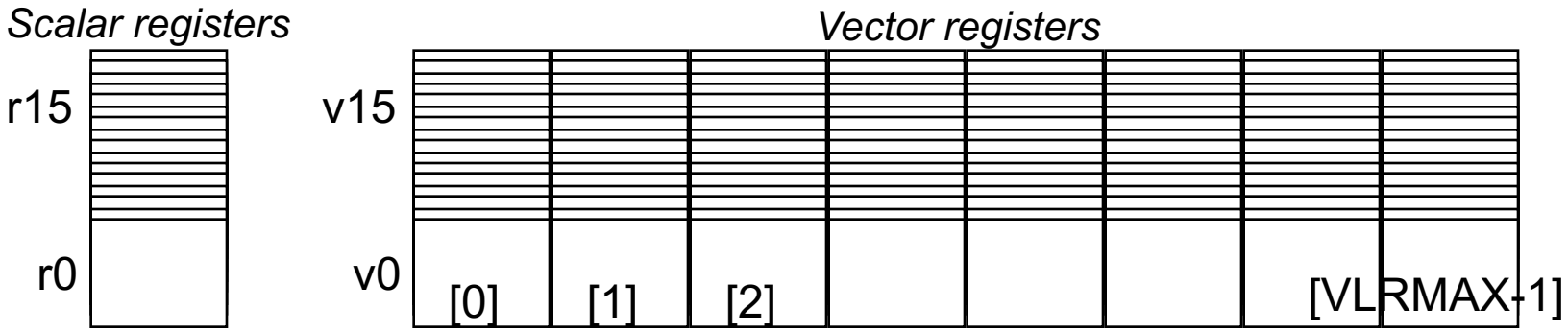
Cray-1 (1976)

Cray-1 (1976)



Memory bank cycle 50 ns *Processor cycle* 12.5 ns (80MHz)

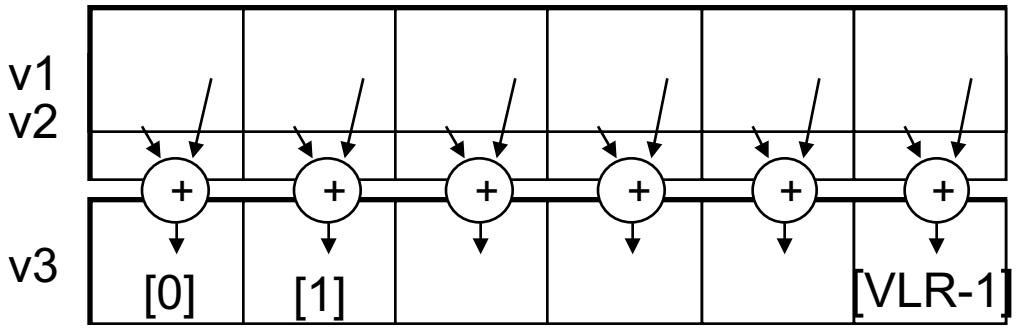
Vector Programming Model



Vector length register VLR

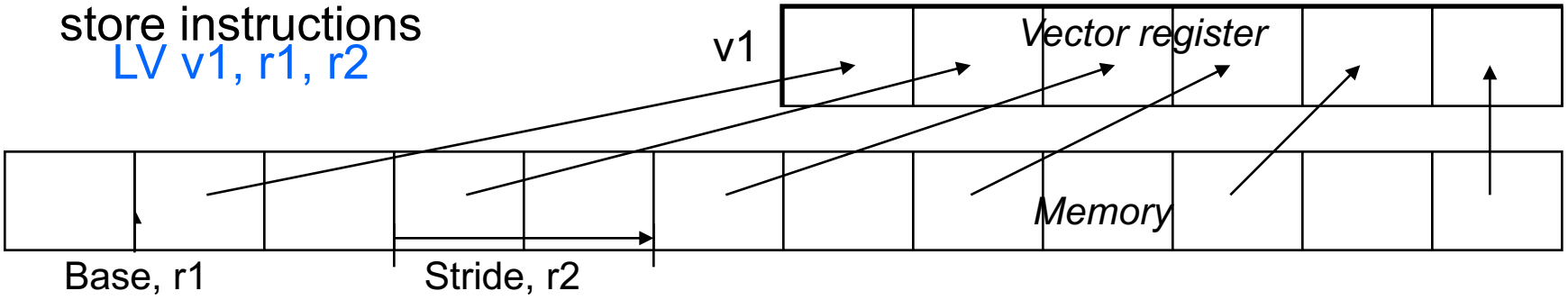
Vector arithmetic instructions

ADDV v3, v1, v2



Vector load and store instructions

LV v1, r1, r2



Vector Code Example

C code

```
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

Scalar code

```
   LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

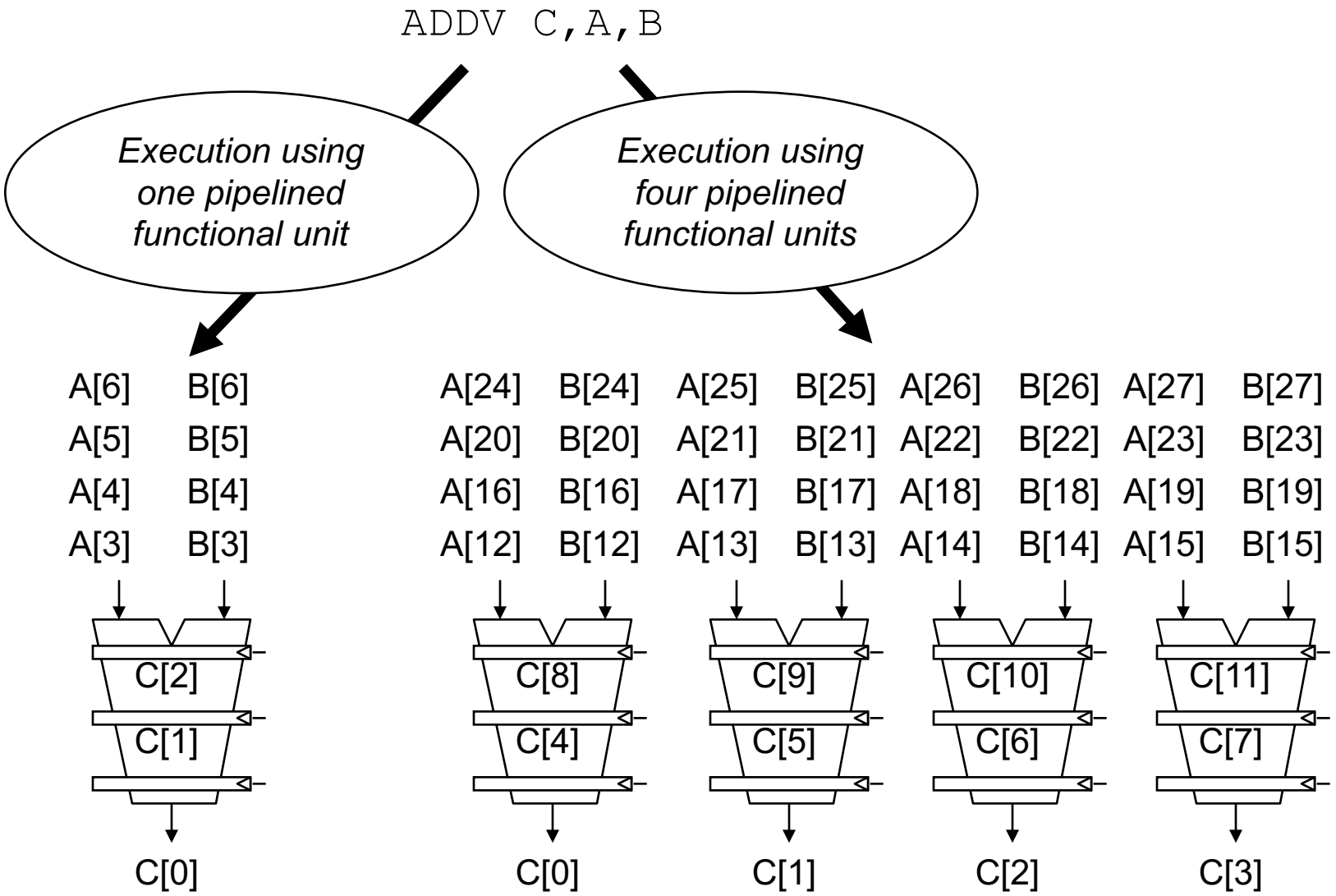
Vector code

```
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

Vector Instruction Set Advantages

- Compact
  - One short instruction encodes N operations.
- Expressive, tells hardware that these N operations:
  - Are independent
  - Use the same functional unit
  - Access disjoint registers
  - Access registers in the same pattern as previous instructions
  - Access a contiguous block of memory (unit-stride load/store)
  - Access memory in a known pattern (strided load/store)
- Scalable
  - Can run same object code on more parallel pipelines or lanes

Vector Instruction Execution



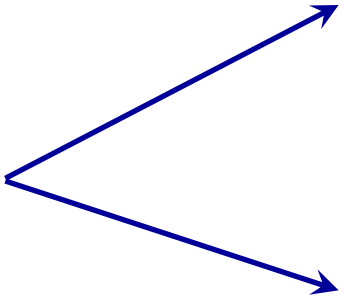
Deep pipelining is possible due to no hazards.

Vector Memory-Memory vs.Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory.
- The first vector machines, CDC Star-100 (1973) and TI ASC (1971), were memory-memory machines.
- Cray-1 (1976) was first vector register machine.

Example source code

```
for (i=0; i<N; i++)
{
  C[i] = A[i] + B[i];
  D[i] = A[i] - B[i];
}
```



Vector memory-memory code

```
ADDV C, A, B
SUBV D, A, B
```

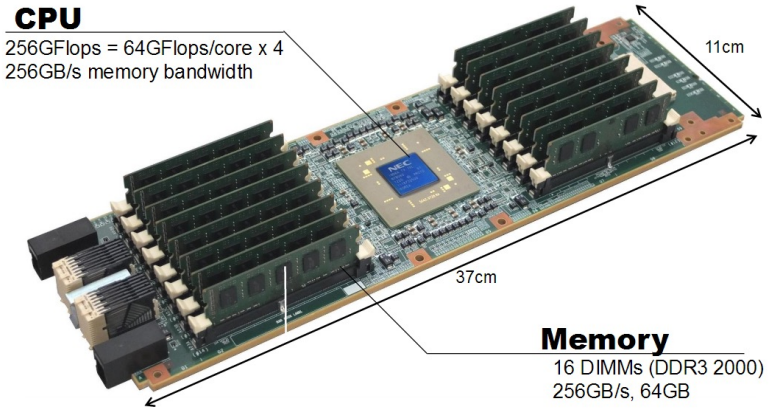
Vector register code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines (cont.)

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth. Why?
  - All operands must be read in and out of memory.
- VMMAs make if difficult to overlap execution of multiple vector operations. Why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency.
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements.
  - For Cray-1, vector/scalar breakeven point was around two elements.
- Apart from CDC follow-ons (Cyber-205, ETA-10), all major vector machines since Cray-1 have had **vector register architectures.**

A Modern Vector Super: NEC SX-ACE (2013)



- Four cores
- 28 nm, 2 B transistors, 1 GHz
- Die size: 23.05 x 24.75 mm,
- SPU decode rate four instructions
- VPU performance 64G flops
- Memory bandwidth 64 GB/s ~ 256 GB/s
- 256 GFlops performance
- 256 GB/s memory bandwidth
- 64 GB memory capacity
- 64 B/128 B memory access granularity (to reduce power)
- 469W rated power consumption

ENGINEERING@SYRACUSE

SIMD Instructions Set Extensions for Multimedia

Multimedia Extensions

- Very short vectors added to existing ISAs for micros.
- Usually 64-bit registers split into 2 × 32 b or 4 × 16 b or 8 × 8 b.
- Newer designs have 128-bit registers (Altivec, SSE2).
- Limited instruction set:
  - No vector length control.
  - No strided load/store or scatter/gather.
  - Unit-stride loads must be aligned to 64/128-bit boundary.
- Limited vector register length:
  - Requires superscalar dispatch to keep multiply/add/load units busy.
  - Loop unrolling to hide latencies increases register pressure.
- Trend toward fuller vector support in microprocessors.

SIMD Extensions

- Media applications operate on data types narrower than the native word size.
  - E.g., disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code.
  - No sophisticated addressing modes (strided, scatter-gather).
  - No mask registers.

SIMD Implementations

- Implementations:
  - Intel MMX (1996).
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD extensions (SSE) (1999).
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit int/fp ops
  - Advanced vector extensions (2010).
    - Four 64-bit int/fp ops
  - Operands must be consecutive and aligned memory locations.

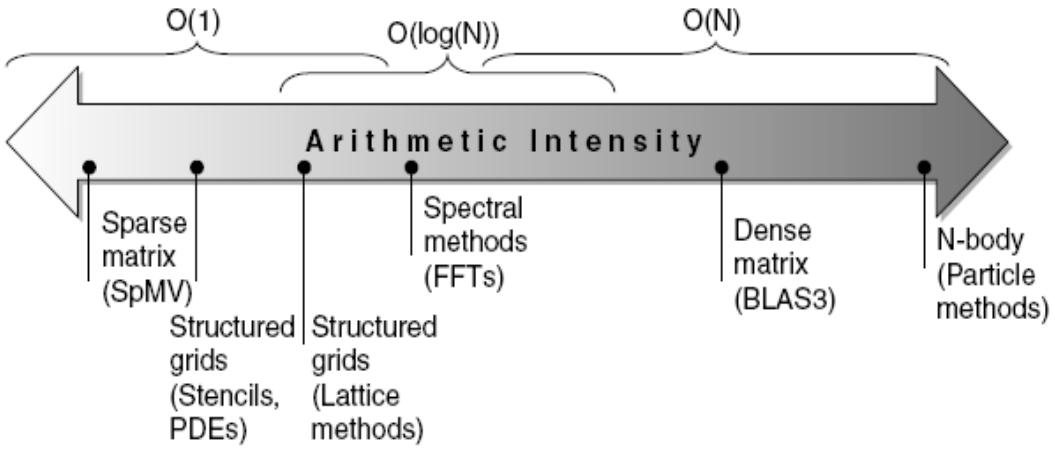
Example SIMD Code

- Example DXPY:

```
  L.D       F0,a          ;load scalar a
  MOV       F1, F0        ;copy a into F1 for SIMD MUL
  MOV       F2, F0        ;copy a into F2 for SIMD MUL
  MOV       F3, F0        ;copy a into F3 for SIMD MUL
  DADDIU    R4,Rx,#512    ;last address to load
Loop:       L.4D F4,0[Rx] ;load X[i], X[i+1], X[i+2], X[i+3]
  MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
  L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
  ADD.4D    F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
  S.4D      0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
  DADDIU    Rx,Rx,#32     ;increment index to X
  DADDIU    Ry,Ry,#32     ;increment index to Y
  DSUBU     R20,R4,Rx     ;compute bound
  BNEZ      R20,Loop      ;check if done
```

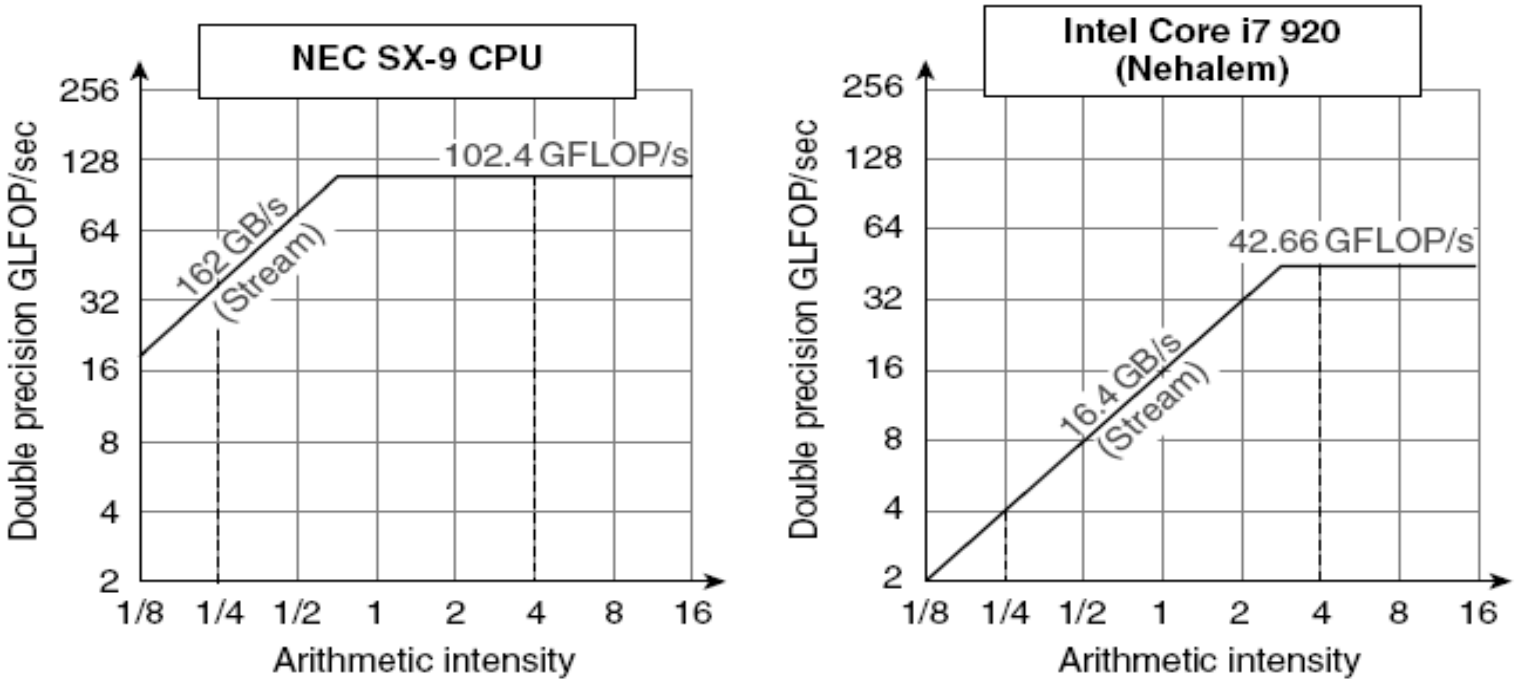
Roofline Performance Model

- Basic idea
  - Plot peak floating-point throughput as a function of arithmetic intensity.
  - Ties together floating-point performance and memory performance for a target machine.
- Arithmetic intensity
  - Floating-point operations per byte



Examples

- Attainable GFLOPs/sec = min (peak memory BW × arithmetic intensity, peak floating point performance)



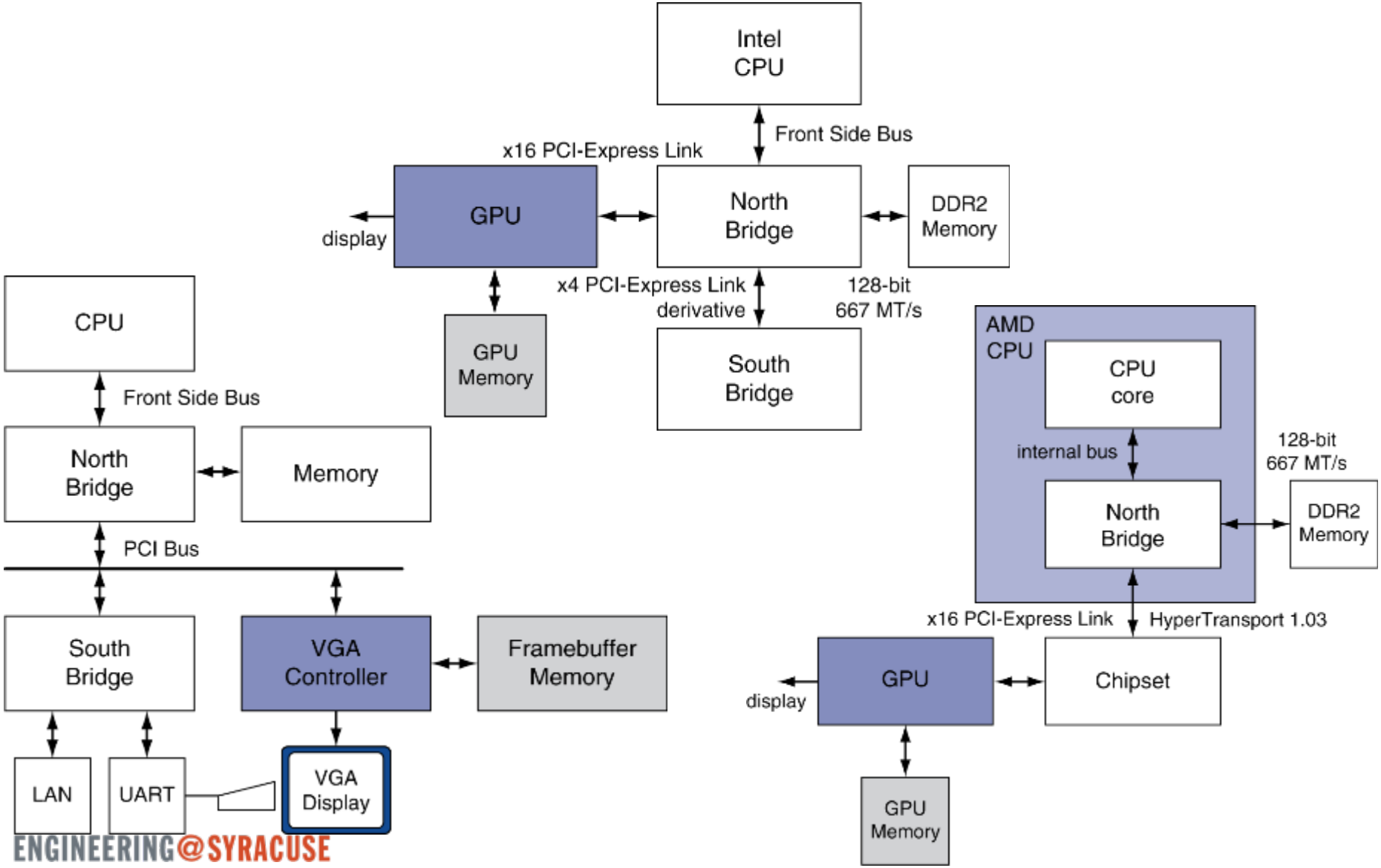
ENGINEERING@SYRACUSE

Graphics Processing Units

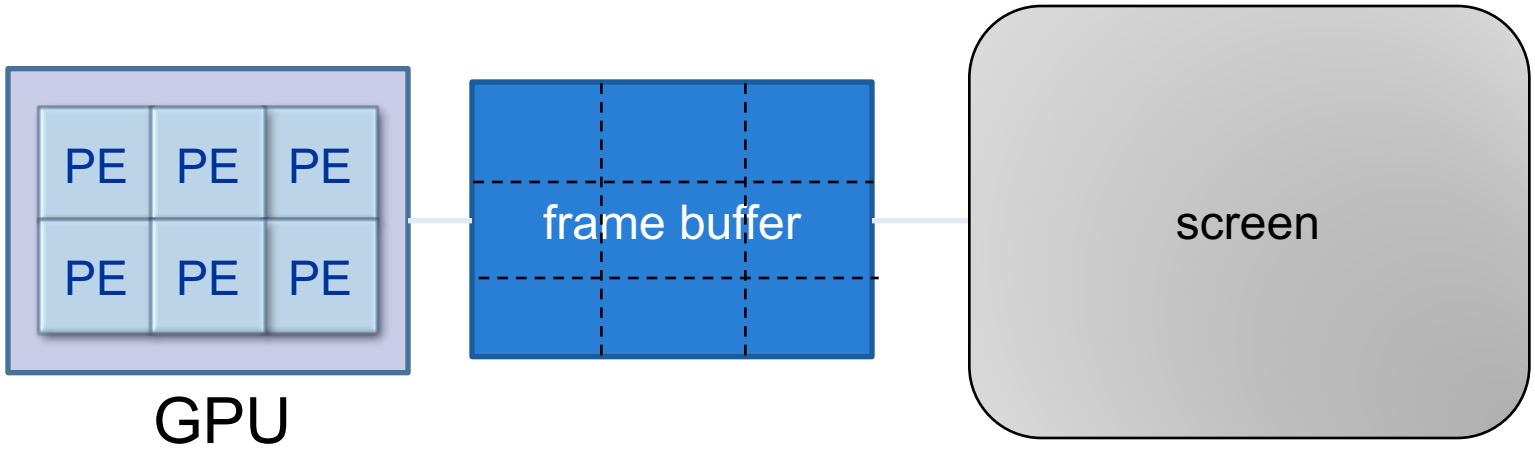
History of GPUs

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

Graphics in the System



GPUs



- Graphics processing units (GPUs) are widely used in desktop/laptop systems and increasingly used in cell phones.
- SIMD architecture maps onto frame buffer.

GPU Architectures

- Processing is highly data-parallel.
  - GPUs are highly multithreaded.
  - Use thread switching to hide memory latency.
    - Less reliance on multilevel 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)

Graphical Processing Units

- Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

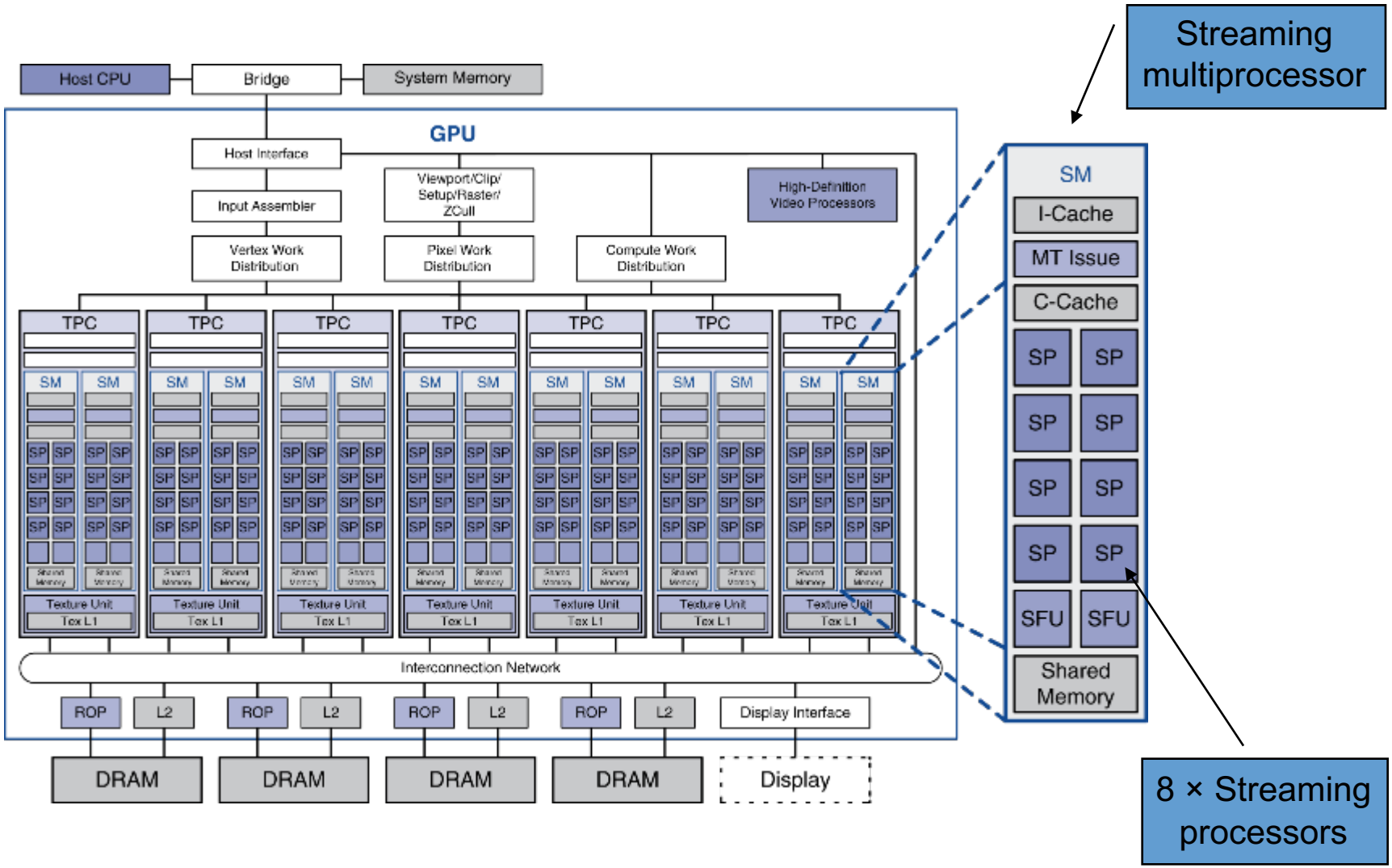
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*; GPU is the *device.*
  - Develop a C-like programming language for GPU.
  - Unify all forms of GPU parallelism as *CUDA thread.*
  - Programming model is “single instruction multiple thread.”

Threads and Blocks

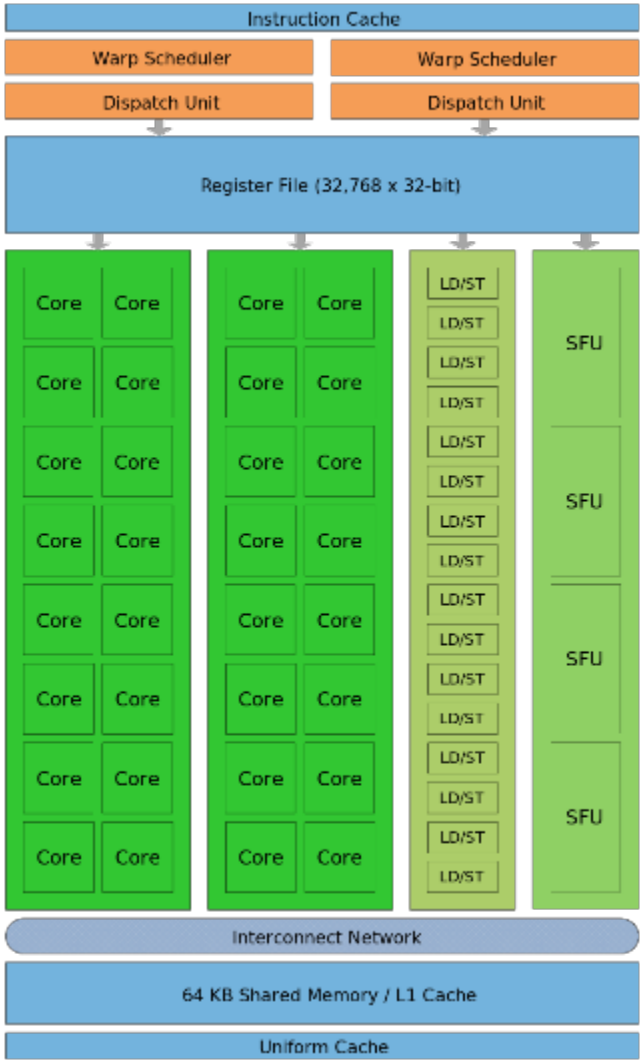
- A thread is associated with each data element.
- Threads are organized into blocks.
- Blocks are organized into a grid.

- GPU hardware handles thread management, not applications or OS.

Example: NVIDIA Tesla



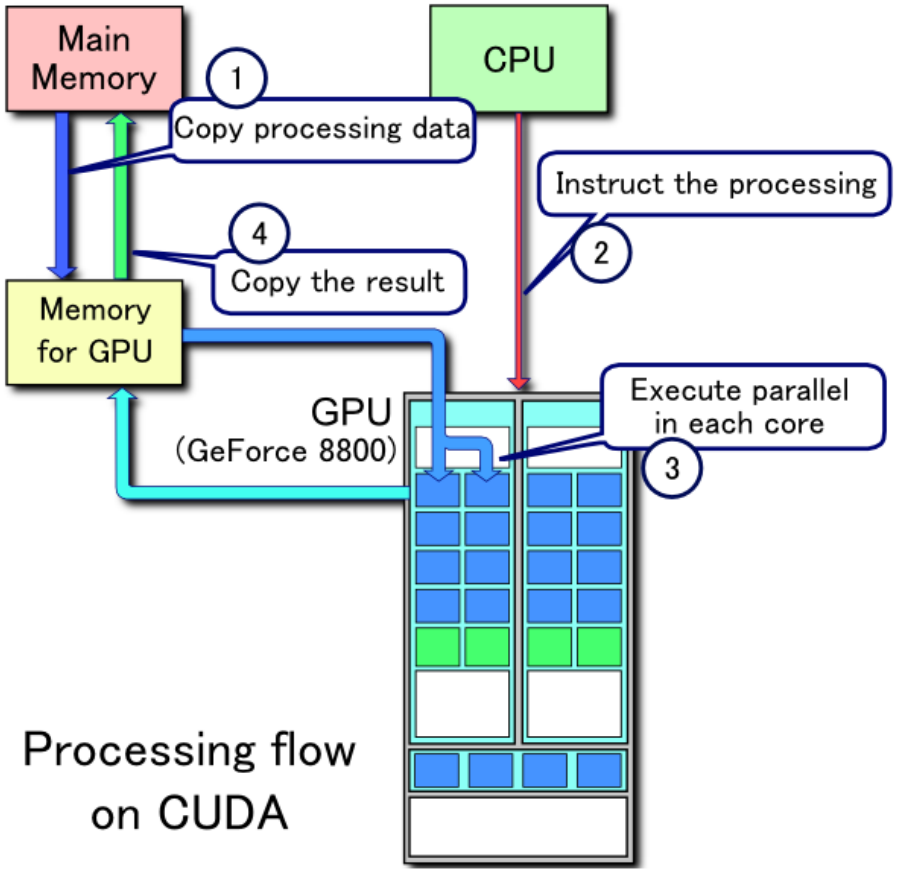
NVIDIA Fermi Architecture



- Three types of processing units:
  - Cores, load/store, special function
  - Controlled by warp scheduler/dispatch
- Warp is group of 32 parallel threads.
- Register file, shared memory, L1, and uniform cache for data.

CUDA Core Architecture

- Thread is most basic unit of programming.
  - Each thread has its own PC, registers, private memory, inputs, and outputs.
- Thread block is set of threads with block ID that shares memory.
- Grid is an array of thread blocks executing on same kernel, sharing global memory.



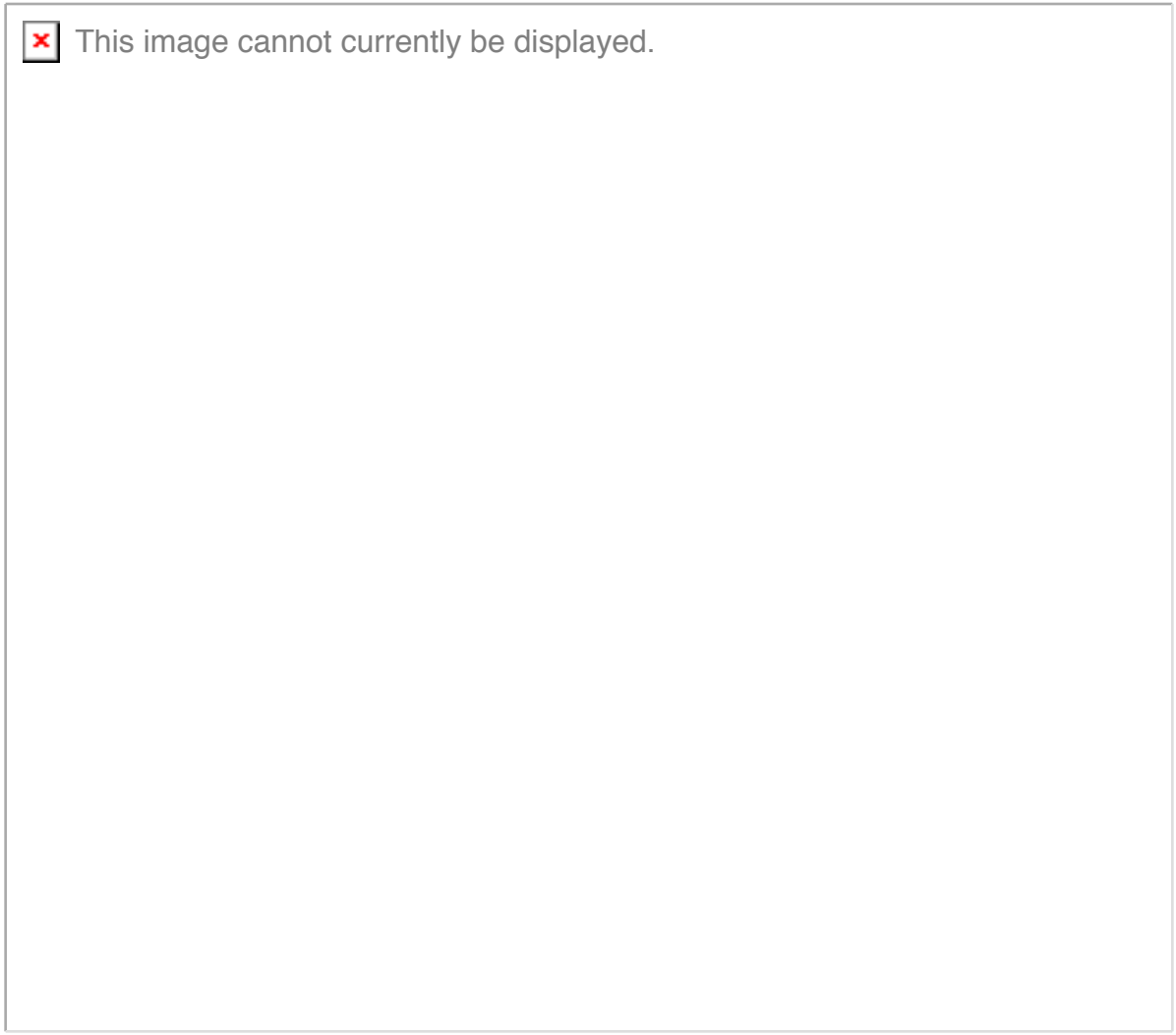
CUDA: *Compute Unified Device Architecture*

Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD.
    - But with performance degradation
    - Need to write general-purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

GPU Memory Structures



Putting GPUs into Perspective

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Guide to GPU Terms

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

ENGINEERING@SYRACUSE

NVIDIA GPU Systems

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Multiply two vectors of length 8192.
  - Code that works over all elements in the grid.
  - Thread blocks break this down into manageable sizes.
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time.
  - Thus grid size = 16 blocks.
  - Block is analogous to a strip-mined vector loop with vector length of 32.
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler.*
  - Current-generation GPUs (Fermi) have seven to 15 multithreaded SIMD processors.

Terminology

- *Threads of SIMD instructions*
  - Each has its own PC.
  - Thread scheduler uses scoreboard to dispatch.
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions.
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors.
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Example

- NVIDIA GPU has 32,768 registers.
  - Divided into lanes.
  - Each SIMD thread is limited to 64 registers.
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers.

NVIDIA Instruction Set Architecture

- ISA is an abstraction of the hardware instruction set.
  - "Parallel thread execution (PTX)."
  - Uses virtual registers.
  - Translation to machine code is performed in software.
  - Example:

```
shl.s32         R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64         R0D, RD0, RD4      ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64         R0D, RD0, RD2      ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64   [Y+R8], RD0        ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks.
- Also uses:
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane.
    - I.e., which threads commit their results (all threads execute).
  - Instruction markers to manage when a branch diverges into multiple execution paths.
    - Push on divergent branch
  - …and when paths converge.
    - Act as barriers
    - Pops stack
- Per-thread-lane one-bit predicate register, specified by programmer.

Example

```
if (X[i] != 0)
  X[i] = X[i] – Y[i];
else X[i] = Z[i];

ld.global.f64  RD0, [X+R8]    ; RD0 = X[i]
setp.neq.s32   P1, RD0, #0    ; P1 is predicate register 1
@!P1, bra      ELSE1, *Push   ; Push old mask, set new mask bits
                              ; if P1 false, go to ELSE1
ld.global.f64  RD2, [Y+R8]    ; RD2 = Y[i]
sub.f64        RD0, RD0, RD2  ; Difference in RD0
st.global.f64  [X+R8], RD0    ; X[i] = RD0
@P1, bra       ENDIF1, *Comp  ; complement mask bits
                              ; if P1 true, go to ENDIF1
ELSE1: ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
       st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1: <next instruction>, *Pop   ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM.
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory.
  - Shared by SIMD lanes/threads within a block
- Memory shared by SIMD processors is GPU memory.
  - Host can read and write GPU memory.

GPUs and Automobiles

- Infotainment System, Smart Display, Digital Cockpit by NVIDIA Tegra Mobile Processors

ENGINEERING@SYRACUSE

Conclusions

In Conclusion

- ILP → TLP → DLP
  - SIMD: short vectors are processed in parallel
  - SMT: several threads are executed in parallel
  - SIMT: vector processing and hardware threading
- Demand and technology
  - Data-intensive applications & CPU + GP-GPU
- Vector machines and supercomputers
  - Lots of registers, memory banks
  - Good for vector operations
  - Simple programming
- SIMD extensions for multimedia
- GPU systems
  - Faster/efficient executions

ENGINEERING@SYRACUSE