CSC290/420 ML Systems for Efficient AI Compute: SIMD and GPUs

Sreepathi Pai

September 3, 2025

URCS

Outline

SIMD Execution: Vector Processors

SIMD Execution: GPUs

Programming Vector Machines

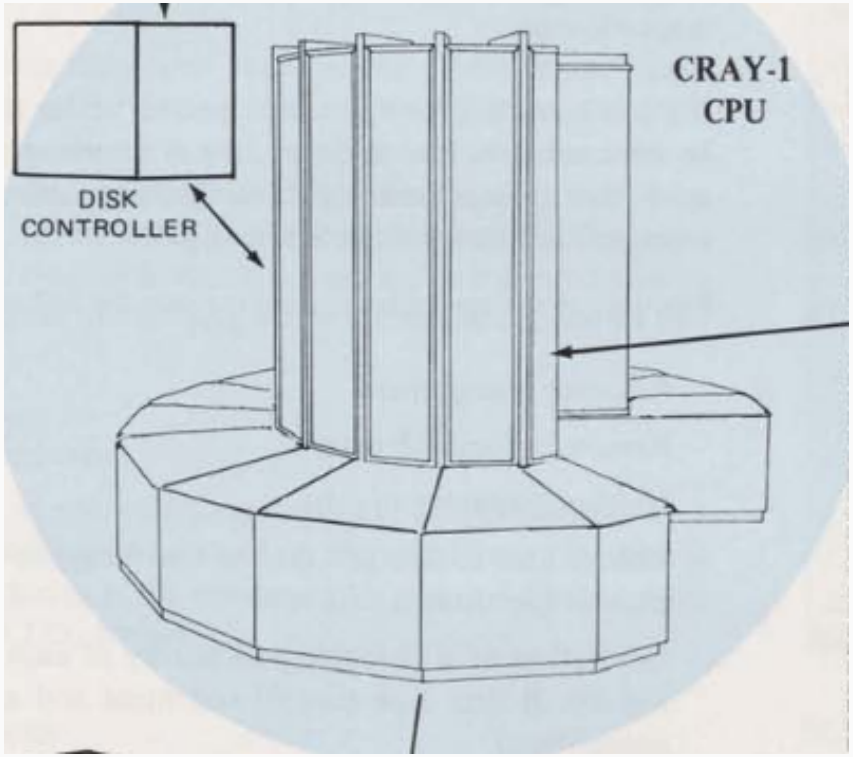
Outline

Vector processing

A vector processor executes vector instructions which operate on on vector registers

- Traditionally associated with super computing
- Still the backbone of (some) modern supercomputers
    - esp. Japanese

From the Cray-1 manual.

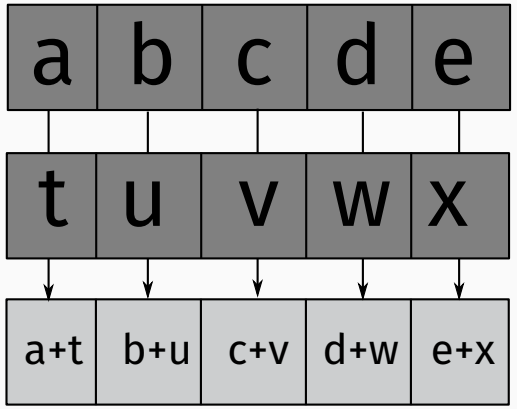


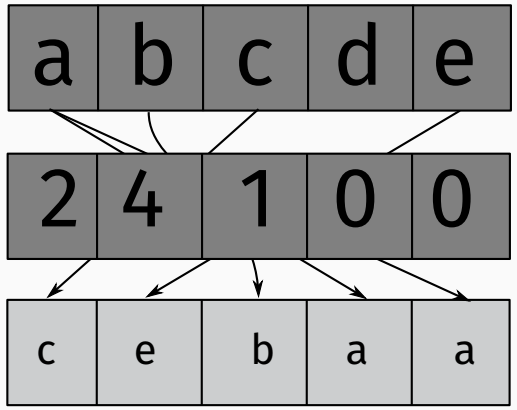
Vector registers

- Scalar register: RAX (x86-64)
  - size: 64 bits
- (short) Vector register: YMM (AVX)
  - size: 256 bits
- On the Cray-1 (1970s)
  - size: 64 elements x 64-bits: 4096 bits

Vector Instructions

- Same operation applied to all elements of the vector
  - subject to predication and masking [next slide]
- "Vertical" vector instructions
  - also called elementwise
  - VC = VA + VB
- "Horizontal" vector instructions
  - operate within a single vector
  - can produce a scalar: A = MAX(VA)
  - can produce a vector: VC = PERMUTE(VA, VB)



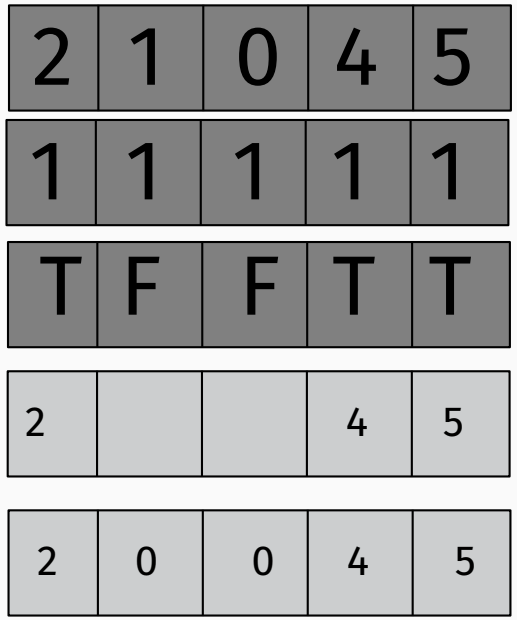


Predication (or Masking)

```
for(i = 0; i < N; i++) {
  if(a[i] > 1)
     c[i] = a[i]
  else
     c[i] = 0
}
```

can be translated into a fictional vector instruction set as (assuming N is the same as the vector size):

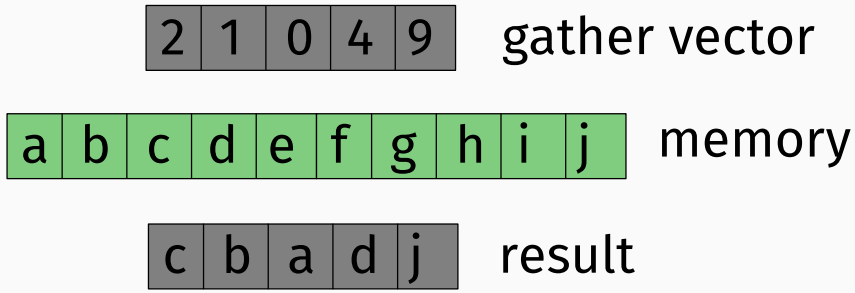
```
      vp = va > v1
@vp   vc = va
@!vp  vc = v0
```



Loads / Stores

- Loading data into vector registers
  - Vector loads: start address + length
  - Load contiguous memory locations
- Storing data into memory
  - Vector stores: destination address + length
  - Stores into contiguous memory locations
- Addresses are usually *aligned*
  - More on this in data structures and memory
- More powerful: vector gathers and scatters

Gathers / Scatters



- Indirect loads (Gathers)
- Indirect stores (Scatters)
- One vector register contains addresses to load to (or store to)
- The Gather and Scatter instructions load the data from these locations
- Convenient, but can affect performance severely

Short Vectors

- SIMD programming on most CPUs
- Much shorter vector sizes
  - Rarely above 512 bits, more commonly 256 bits
- Limited set of operations
- Examples: x86 MMX/SSE/.../AVX/AVX2/AVX512, PowerPC AltiVec, ARM NEON
  - x86-64 quirk: Only SSE supports IEEE 754

Programming Vector Processors

- Explicitly parallel
  - Programmer writes code using vector instructions
  - Sometimes called SIMD intrinsics
  - CPU-specific
- Implicitly parallel
  - Compiler (not hardware) extracts parallelism from “serial” code
  - Autovectorization

Autovectorization

```
void vec_add(int *A, int *B, int *C, int N) {
  for(int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
gcc -O3 vecadd.c
```

```
        ...
        vmovdqu (%rdi,%rax), %ymm1
        vpaddd  (%rsi,%rax), %ymm1, %ymm0
        vmovdqu %ymm0, (%rdx,%rax)
        ...
```

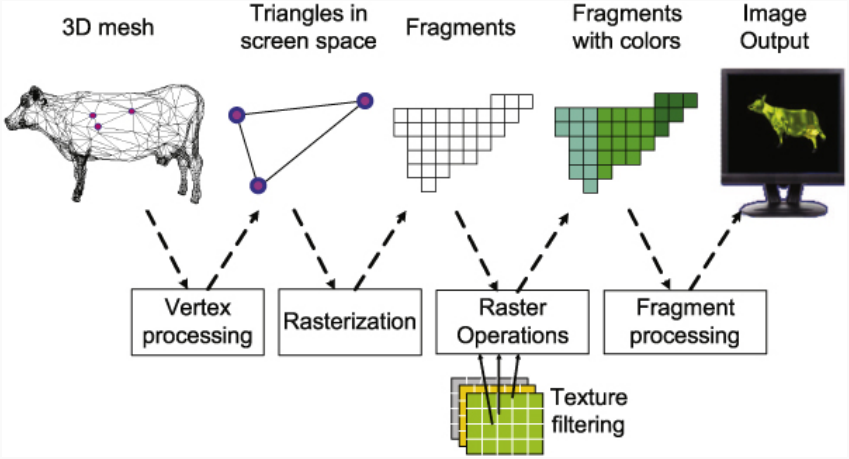
Short vector code

```
__m256i avec0 = load(A + lda * (ii + 0) + l);
__m256i avec1 = load(A + lda * (ii + 1) + l);
__m256i avec2 = load(A + lda * (ii + 2) + l);
__m256i avec3 = load(A + lda * (ii + 3) + l);
for (int64_t j = 0; j < RN; ++j) {
    __m128 db = _mm_set1_ps(unhalf(B[ldb * (jj + j) + l].d));
    // Computation of product of delta values for four blocks and r
    __m256 dvec =  _mm256_castps128_ps256(_mm_mul_ps(da, db));
    dvec = _mm256_permute2f128_ps(dvec ,dvec, 0);
    // Computation of dot product and multiplication with appropria
    Cv[j][0] = madd(_mm256_shuffle_ps(dvec, dvec, 0),
                    updot(_mm256_sign_epi8(avec0, avec0),
                          _mm256_sign_epi8(load(B + ldb * (jj + j)
                    Cv[j][0]);
```

from https://github.com/ggml-org/ggml/blob/83835ffaa0f2e68bc8530bd0a7584711789dc23b/src/ggml-cpu/ops.cpp

Outline

Graphics Processors (Originally)

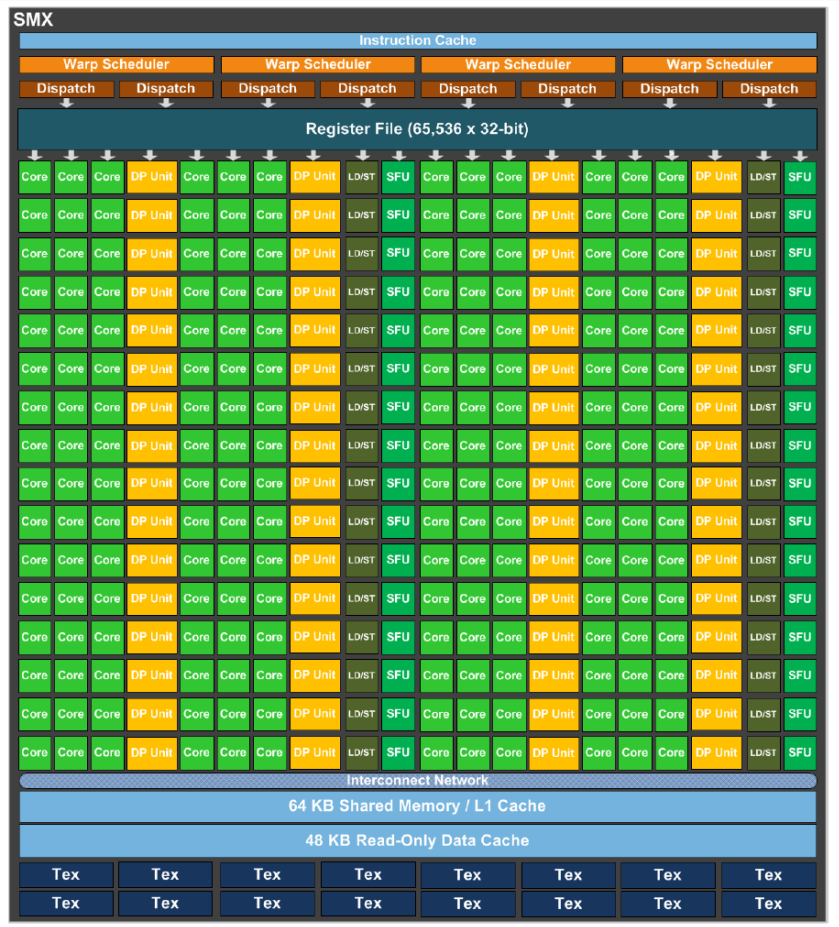


- Built for graphics and 3D games
- Displaying 3D graphics is very compute intensive
  - But also highly parallel
- Standardized "graphics" pipeline
  - Originally dedicated hardware
  - Around 2006, replaced with "unified" general-purpose cores

Kaufman et al (2009)

A modern GPU

- All vector instructions
  - usually 32 element vectors (1024 bits)
- Multiple cores
- Multiple issue but in-order
- Highly SMT
  - 64-way
  - When one thread is waiting, another ready thread starts executing
- Optimized to run thousands of threads
  - Note: each thread runs a vector instruction



Kepler SMX, from the CUDA documentation.

A Kepler GPU (circa 2012)

CUDA Programming Model

- Scalar programming model
  - but compiler *does not* autovectorize!
- Hardware executes all code in vector mode
  - branches are handled by hardware
  - all loads/stores are gather/scatter
- Lanes of vectors are exposed to programmers as [CUDA] "threads"
  - Unlike CPU threads, CUDA threads don't have an independent program counter
  - But hardware makes them appear like CPU threads

Predication / Branch Divergence

- Like vector processors, GPUs also support predication
- But also support "warp divergence" mechanism
  - A "warp" is CUDA terminology for a vector instruction
- In warp divergence, the warp splits into two parts
  - One part executes the true branch (while other part is disabled)
  - And then, another part executes the false branch
  - Both parts "join up" at a pre-determined point
- Predication or warp divergence selected by the compiler on a per-branch basis

Gathers / Scatters

- GPU loads are automatically gathers/scatters
- Best performance if contiguous data accessed
  - "Coalesced" access, similar to vector loads
- Worst performance if every lane accesses a different address(*)
  - *actually cache line

Vector Addition

```
__global__ void vector_addition(int *A, int *B, int *C, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if(tid < N)
       C[tid] = A[tid] + B[tid];
}

vector_addition<<<(N+255)/256, 256>>>(...);
```

Data Parallel Programming

- All data is processed by the same code
  - i.e. every thread runs the same code
- Code decides which data to operate on based on some sort of a "ID"
  - lane ID
  - thread ID

The CUDA Hierarchical Programming Model

- To handle different models of GPUs, CUDA does not use a "flat" identifier space.
- Threads are divided into equal-sized 3-D Thread Blocks
  - All threads of a thread block are guaranteed to be resident on a core
  - Different thread blocks may or may not be running at same time
- Thread blocks form a 3-D Grid
- Total threads is number of thread blocks multiplied by dimensions of thread block
- Other GPU programming models (OpenCL, WebGPU) are similar
  - but use different terminology

Considerations for using GPUs

- Built for *throughput* computing
  - Not latency
  - Do lots of work quickly
  - Can't do little work quickly (remember, in-order!)
- Need lots of work
  - 2048 CUDA lanes per core * 80 cores, for example
- Need to transfer data to and fro from CPU
  - Overhead may be significant and destroy gains from compute

Programming CPUs using GPU programming models?

- The CUDA programming model programs vector machines *without* using vectors
  - and without relying on compiler wizardry
- Intel Implicit SPMD Compiler (ISPC) is a similar model, but for CPUs
  - Not just Intel CPUs
- Slang, is a new-ish shader language
  - Supports both CPUs and GPUs

Outline

It's the **FLOPS!**

Achieving peak compute performance on any modern computer *requires* the use of vector instructions.

Considerations

- Design so code can run on both GPUs and CPUs
  - start with a GPU-first design
- Domain-Specific Languages
  - avoid using low-level GPU/CPU-specific programming languages
  - use high-level domain-specific languages (e.g., Triton)
  - easier and more productive to get performance
- Don't dismiss auto-vectorization for ML
  - IREE