CSC266 Introduction to Parallel Computing using GPUs
Parallelizing Programs

Sreepathi Pai

September 20, 2017

URCS

Outline

Outline

Serial vs Parallel Algorithms

What makes some algorithms serial and others parallel?

Data Dependences

Operation waits for data to become available:

```
a = b + c
```

which gets compiled down to:

```
load r1, b
load r2, c
add r3, r1, r2
```

Although `add` can start before the two `loads`, it has to wait for the `loads` to finish.

Control Dependences

- Operation may or may not execute depending on condition

```
if(temperature == cold)
   wear_jacket()
```

- Control dependences can always be converted to data dependences

Dynamic vs Static Dependences

When can the two writes in the code below execute in parallel?

```
void set_p(int *x, int *y) {
  *x = 1;
  *y = 2;
}
```

When can they not?

Dependence Graphs

- A graphical representation of dependences between operations
  - Data-flow graphs
  - Control-flow graphs
- We will focus on data-flow graphs
  - Nodes: operations in *dynamic trace*
  - Edges: communication, "flow of data"
- Dynamic trace
  - All operations executed by a program other than control flow

Dynamic Trace

Assume a, b and c are not aliases (i.e. they are separate arrays) and $N = 3$.

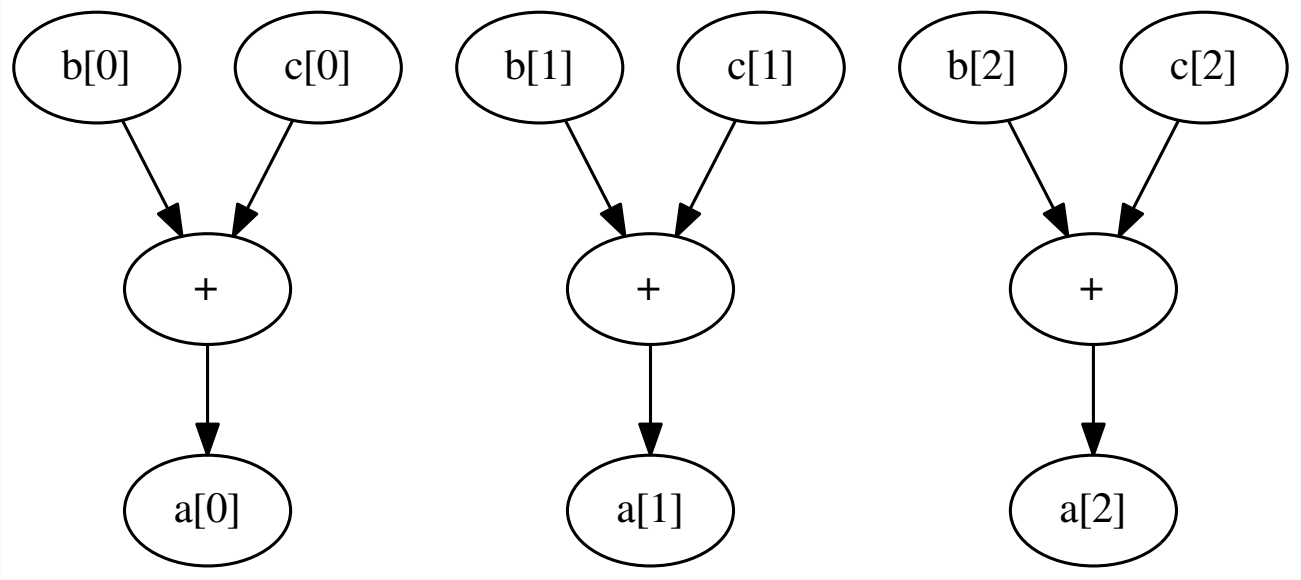
```
for(i = 0; i < N; i++)
   a[i] = b[i] + c[i]
```

The dynamic trace is:

```
a[0] = b[0] + c[0]
a[1] = b[1] + c[1]
a[2] = b[2] + c[2]
```

Drawing a Dependence Graph

```
a[0] = b[0] + c[0]
a[1] = b[1] + c[1]
a[2] = b[2] + c[2]
```



Outline

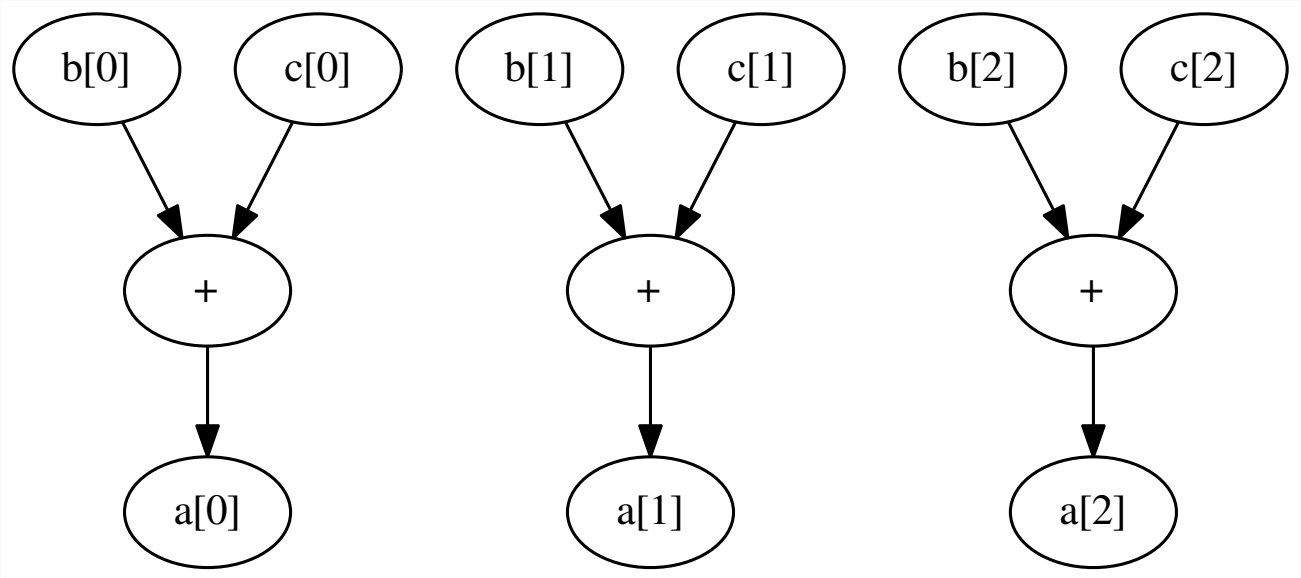
Vector Addition

Assume a, b and c all arrays of length N.

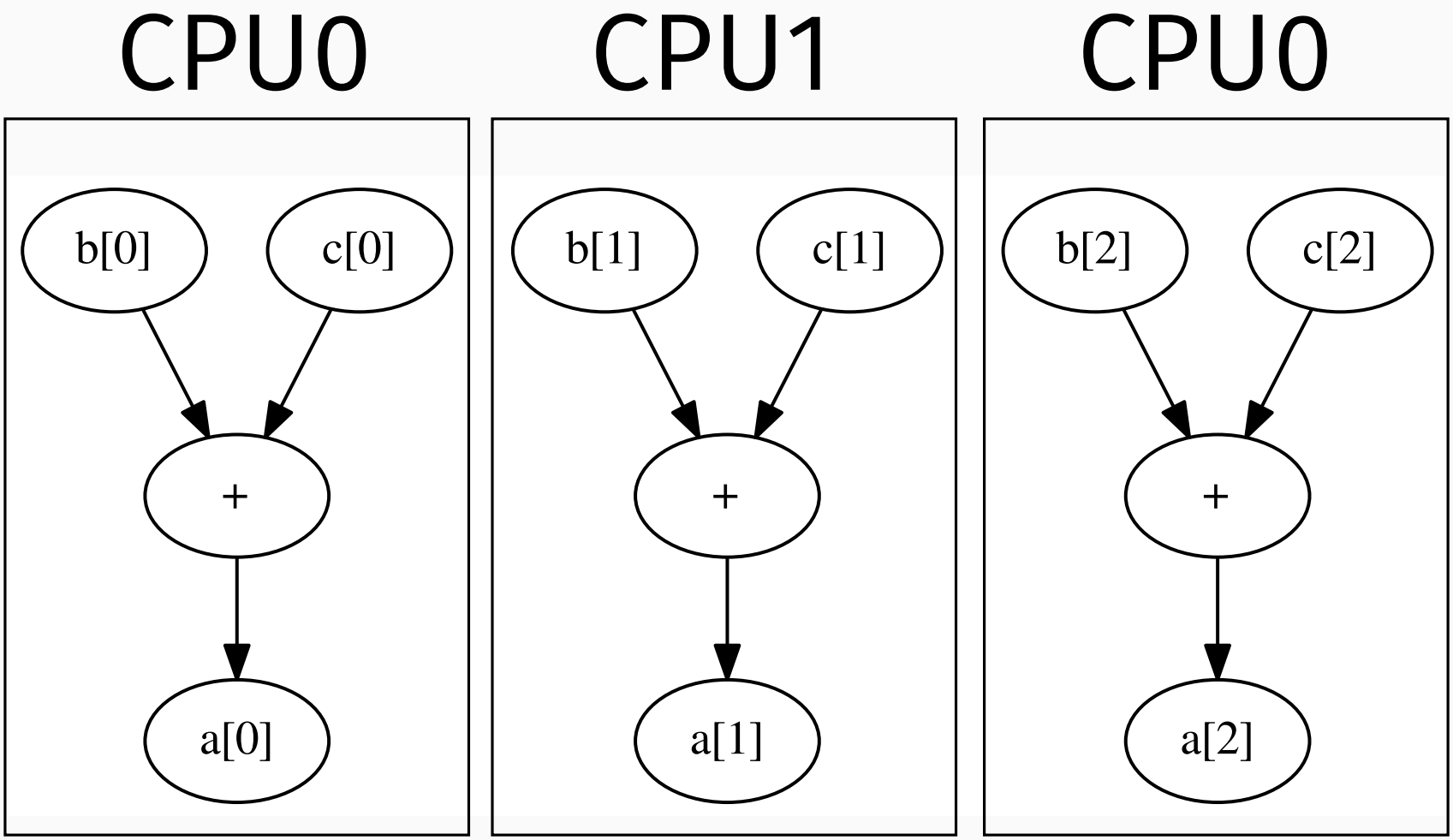
```
void vadd(a, b, c, N) {
   for(i = 0; i < N; i++)
      a[i] = b[i] + c[i]
}
```

Does iteration 1 of this loop depend on iteration 0 of this loop?

Dependence Graph for Vector Addition



Distributing Work for Vector Addition



Sometimes called "embarrassingly parallel" – no communication between parallel computations

Vector Sum

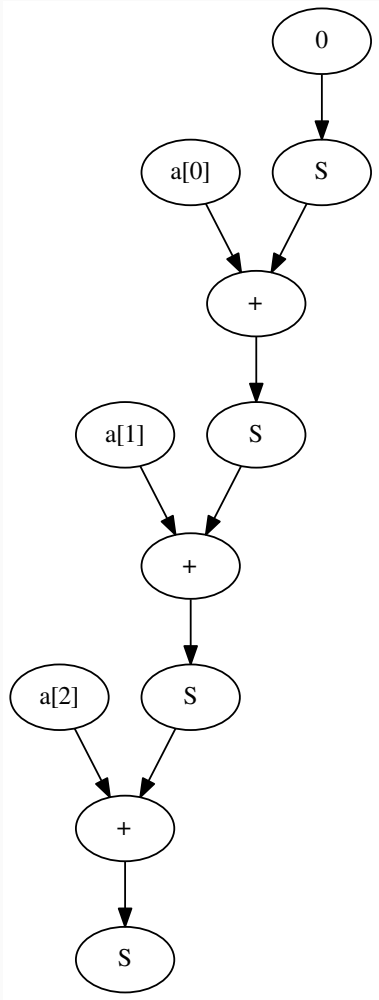
Assume a is array of length $N = 3$.

```
void vsum(a) {
    S = 0
    for(i = 0; i < N; i++)
       S = S + a[i]
}
```

What does the dependence graph for this loop look like?

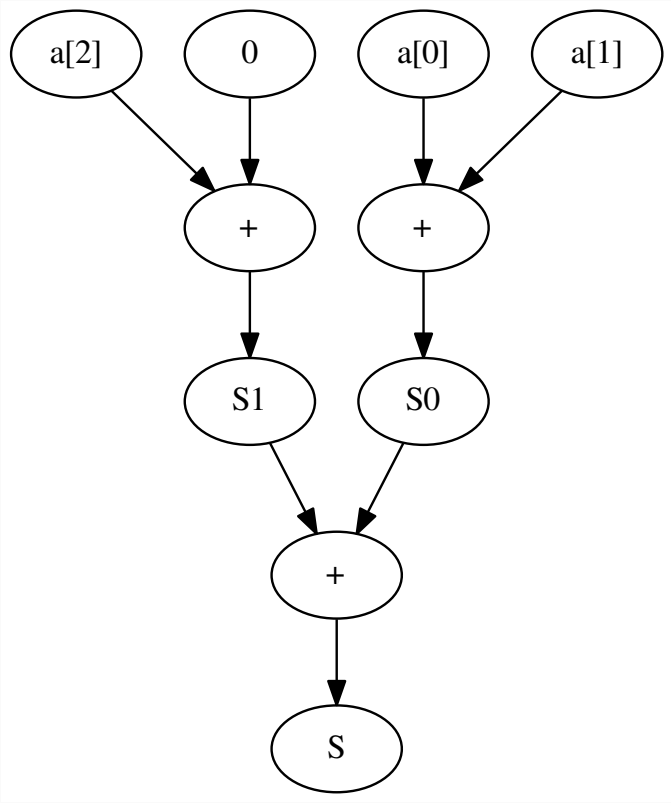
Dependence Graph for Vector Sum

```
S = 0
S = S + A[0]
S = S + A[1]
S = S + A[2]
```

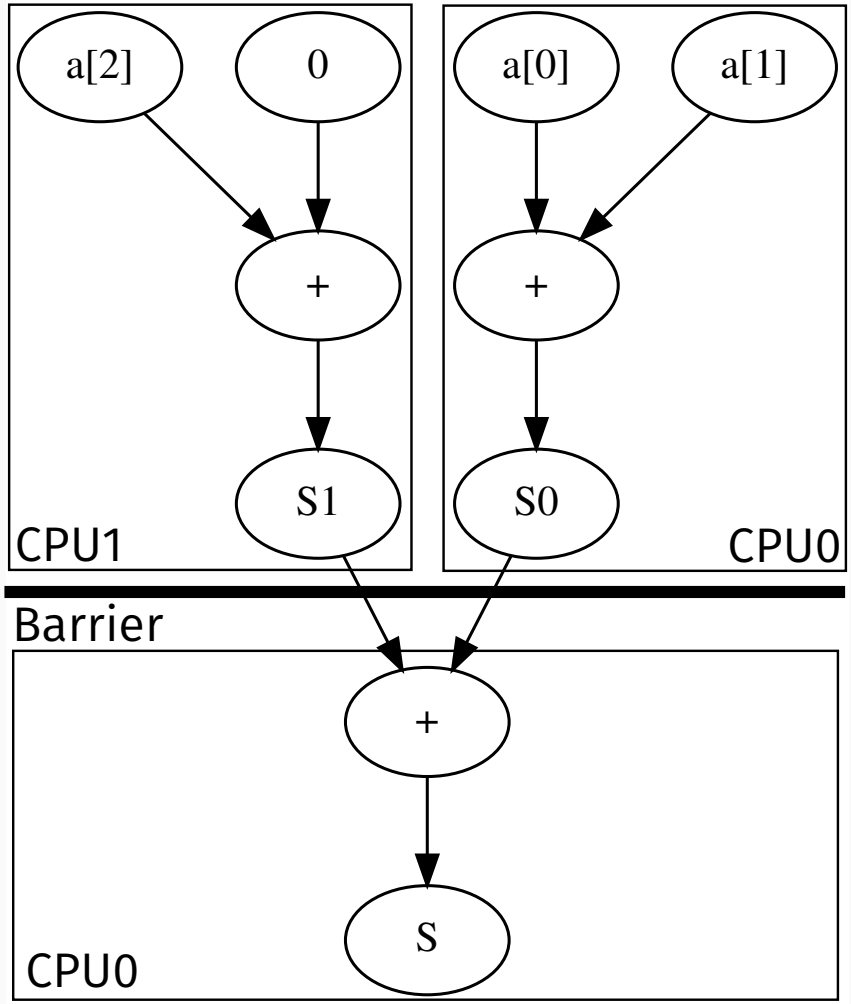


Breaking Dependences for Vector Sum: Reduction Tree

- Create an array of S
  - One element for each thread
- Get sum of each thread as an array
  - S[0..thread]
- Repeat on this array until only one element left
- Requires synchronization



Distributing Work for Vector Sum



Nested Parallelism: Matrix Multiplication

- Multiply matrices A and B to get C
  - A is $M \times N$
  - B is $N \times K$
  - C is $M \times K$

```
for(row = 0; row < M; row++)
   for(col = 0; col < K; col++)
      vmul(A[row][:], B[:][col], Tmp, N)
      C[row][col] = vsum(Tmp, N)
```

Irregular Parallelism: Breadth First Search

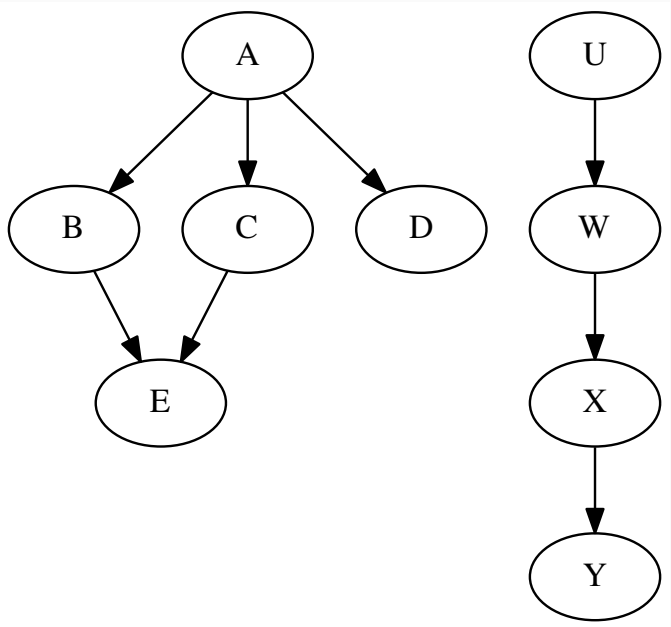
Which of the loops below can be safely parallelized?

```
void bfs(graph, LEVEL) {
   for(n = 0; n < graph.nodes; n++)
      for(e = n.firstedge(); e < n.lastedge(); e++)
         if(e.dst.distance == INF && n.distance == LEVEL - 1)
            e.dst.distance = LEVEL;
}

set all distances to INF
src.distance = 0
LEVEL = 1
do {
   bfs(graph, LEVEL)
   LEVEL++
} while(some edge's distance changed)
```

BFS on two graphs

Does parallelism depend on input?



Outline

Flynn's Taxonomy

How computers are implemented:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

Programmer's Viewpoint

- Task Parallelism
- Data Parallelism

Task Parallelism

- Different code (usually all running at same time)
- Different data
- Example (real life)
  - Assembly line
- Example (CS)
  - Unix pipe invocation: `sort | uniq | wc -l`

Data Parallelism

- Same code
- Runs on different data
- Two important levels
  - Instruction-level (SIMD)
  - Program-level (SPMD)

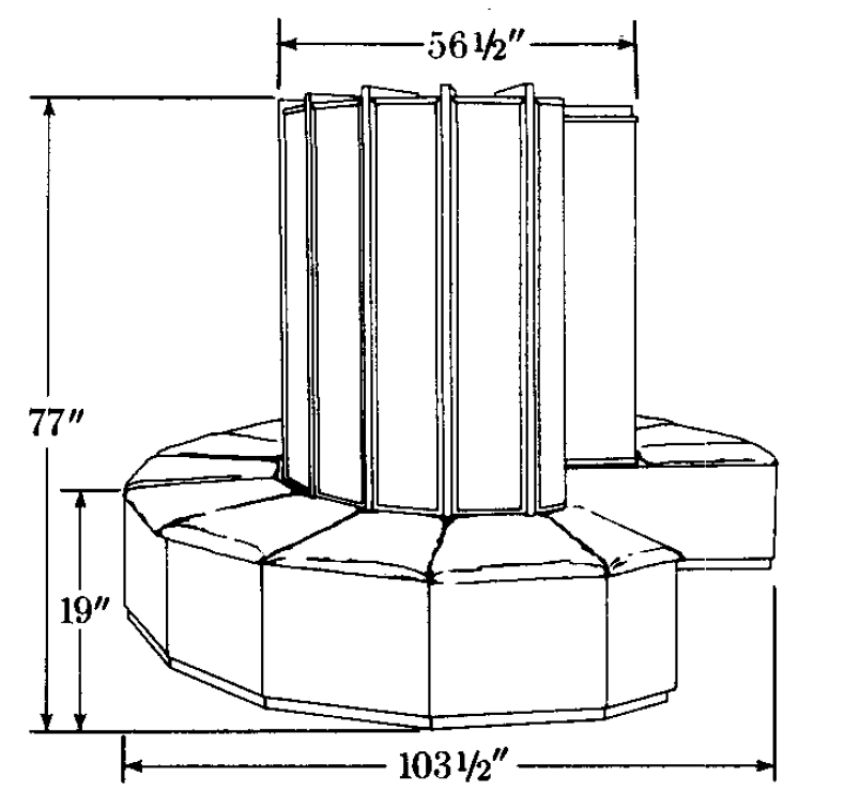
GPUs prefer data parallelism.

SIMD Implementations: Vector Machines

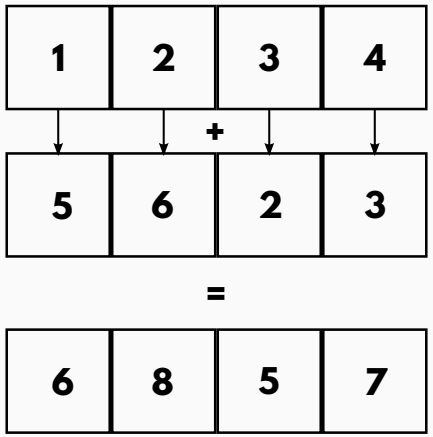
The Cray-1 (circa 1977):

- V*x* – vector registers
  - 64 elements
  - 64-bits per element
- Vector length register (V*len*)
- Vector mask register

Richard Russell, "The Cray-1 Computer System", Comm. ACM 21,1 (Jan 1978), 63-72



Vector Instructions – Vertical

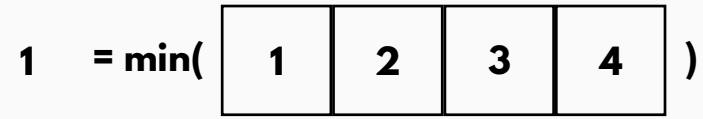


For $0 < i < \mathrm{V}len$:

```
dst[i] = src1[i] + src2[i]
```

- Most arithmetic instructions

Vector Instructions – Horizontal



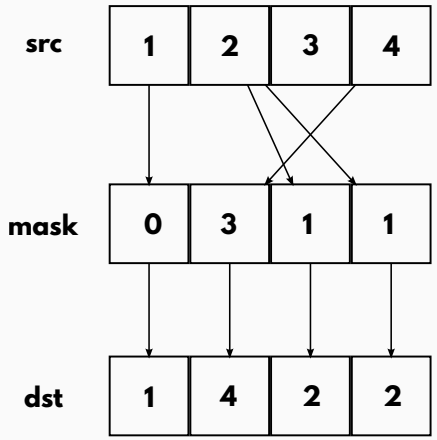
For $0 < i < \mathrm{V}len$:

```
dst = min(src1[i], dst)
```

Note that `dst` is a scalar.

- Mostly reductions (min, max, sum, etc.)
- Not well supported
    - Cray-1 did not have this

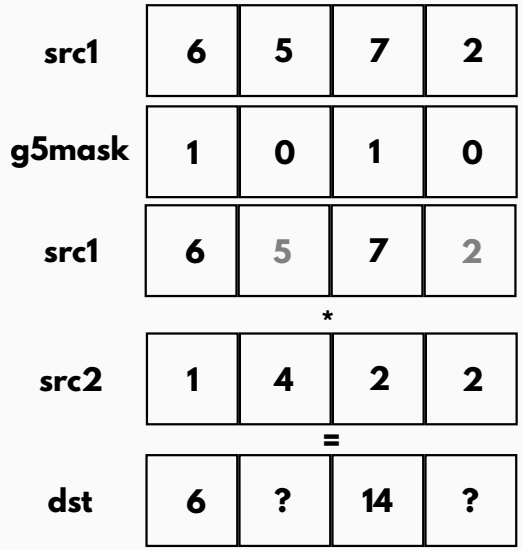
Vector Instructions – Shuffle/Permute




```
dst = shuffle(src1, mask)
```

- Poor support on older implementations
- Reasonably well-supported on recent implementations

Masking/Predication




```
g5mask = gt(src1, 5)
dst = mul(src1, src2, g5mask)
```

Outline

SIMD Registers in modern CPUs

- Bits, not elements
  - 64 bits (MMX, 3DNow!)
  - 128 bits (SSE family, AltiVec)
  - 256 bits (AVX-256)
  - 512 bits (AVX 2.0 or AVX-512)
- Example 128-bit vector can contain
  - 16 8-bit elements
  - 8 16-bit elements
  - 4 32-bit elements
  - 2 64-bit elements
  - 1 128-bit element
- No vector length register
  - except AVX512 which has vector length (but can only choose 128, 256, 512)
- No vector masks
  - except AVX512 which has masks

Four ways to write SIMD programs on current CPUs

In order of decreasing difficulty:

- Assembly code
  - not covering this
- SIMD intrinsics
- Vector types
- Autovectorizing compilers
  - not covering this
  - great when it works!

Vector intrinsics (using x86 as an example)

- Expose short vector instructions to programmer
- Not assembly programming, compiler still does:
  - register allocation
  - scheduling
- New data types (like vector types):
  - `__m64` (MMX)
  - `__m128` (SSE)
  - `__m128i` and `__m128d` (SSE2)
- Intrinsics look like functions:
  - `__m128 _mm_add_ss (__m128 A, __m128 B)`
- Allows direct use of machine instructions

Vector types in GCC

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;
```

- Operations on vector types generate vector instructions
  - Most low-level details (e.g. alignment), taken care of by gcc
- Operations supported:
  - `+`, `-`, `*`, `/`, `unary minus`, `^`, `|`, `&`, `~`, `%`
  - this does not include machine-specific instructions

Conclusion

- Dependence analysis is necessary to identify parallelism
- Important types of parallelism "patterns":
  - embarrassingly parallel (vector addition)
  - tree-based reductions
  - nested parallelism
  - irregular parallelism
- For GPUs: focus on SIMD parallelism
  - Abundant in scientific applications, multimedia applications, etc.
  - Usually found in loops