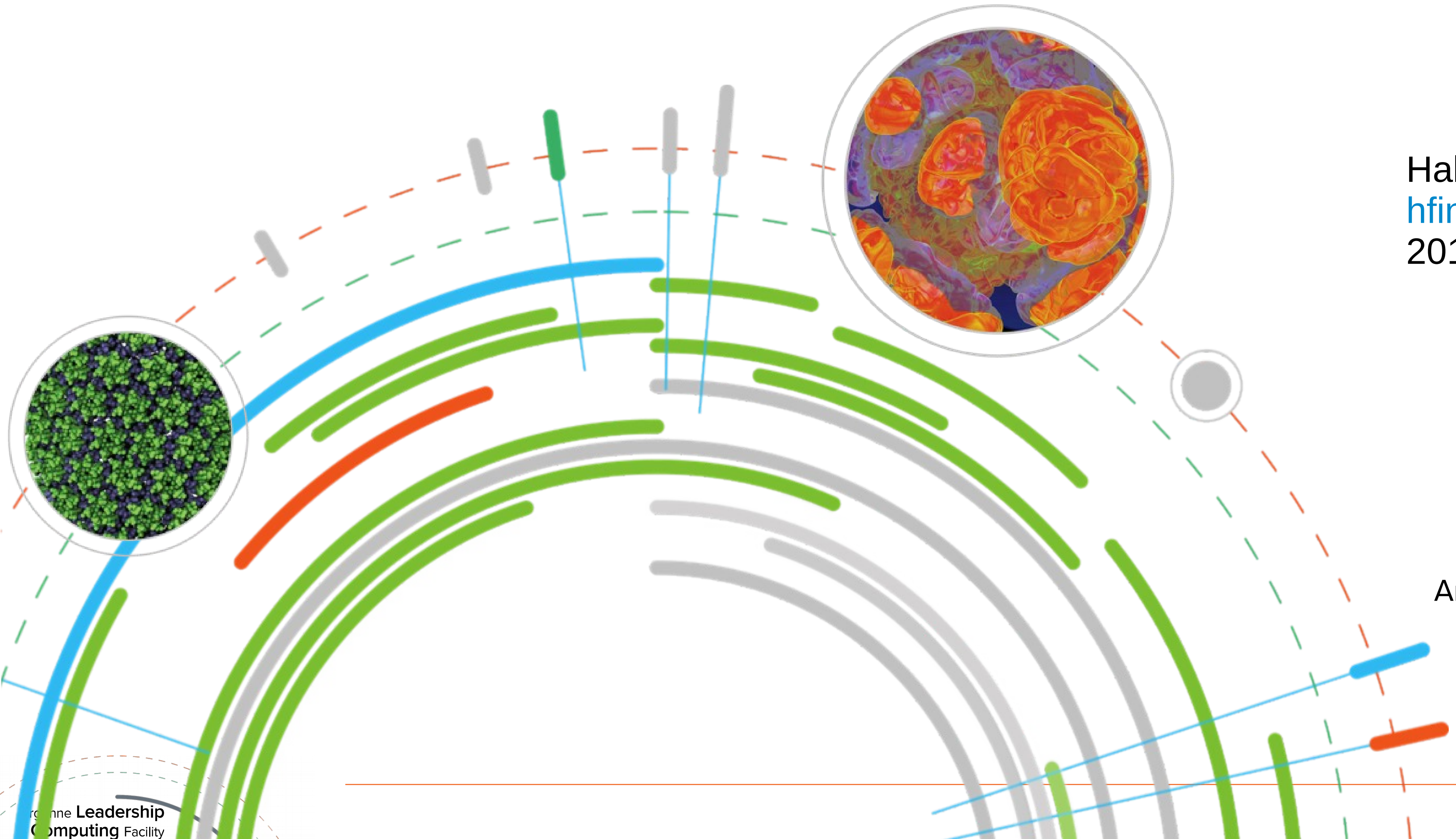
Optimizing for Intel's Knights Landing and Other HPC Architectures

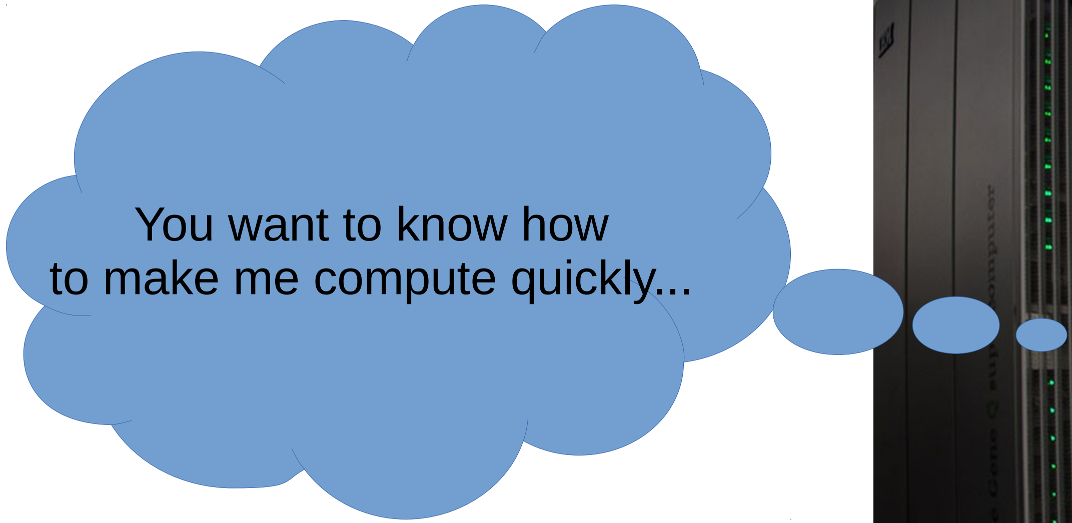


Hal Finkel
hfinkel@anl.gov
2016-07-21

Argonne Leadership Computing Facility

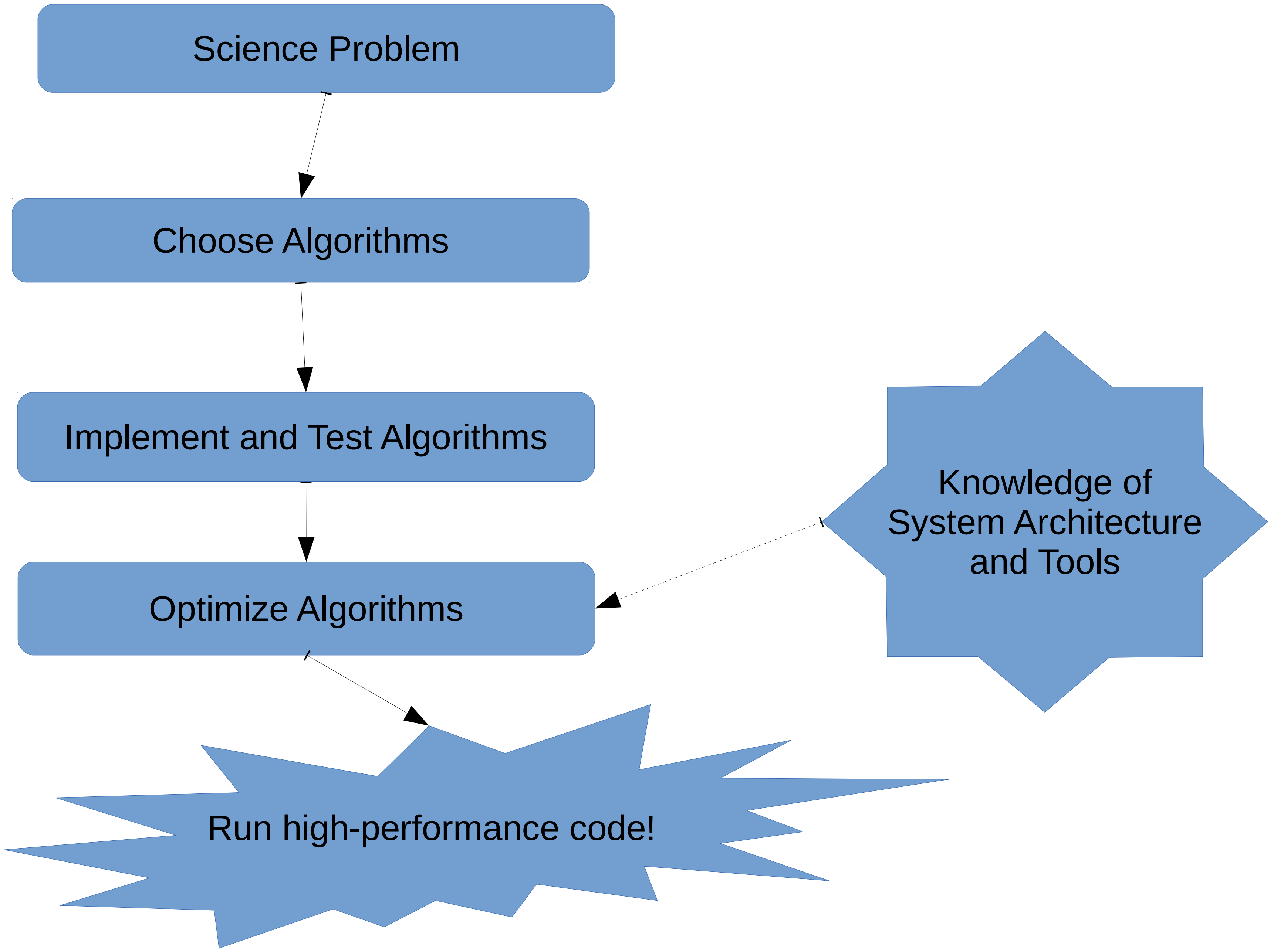
Argonne NATIONAL LABORATORY

Optimizing for HPC

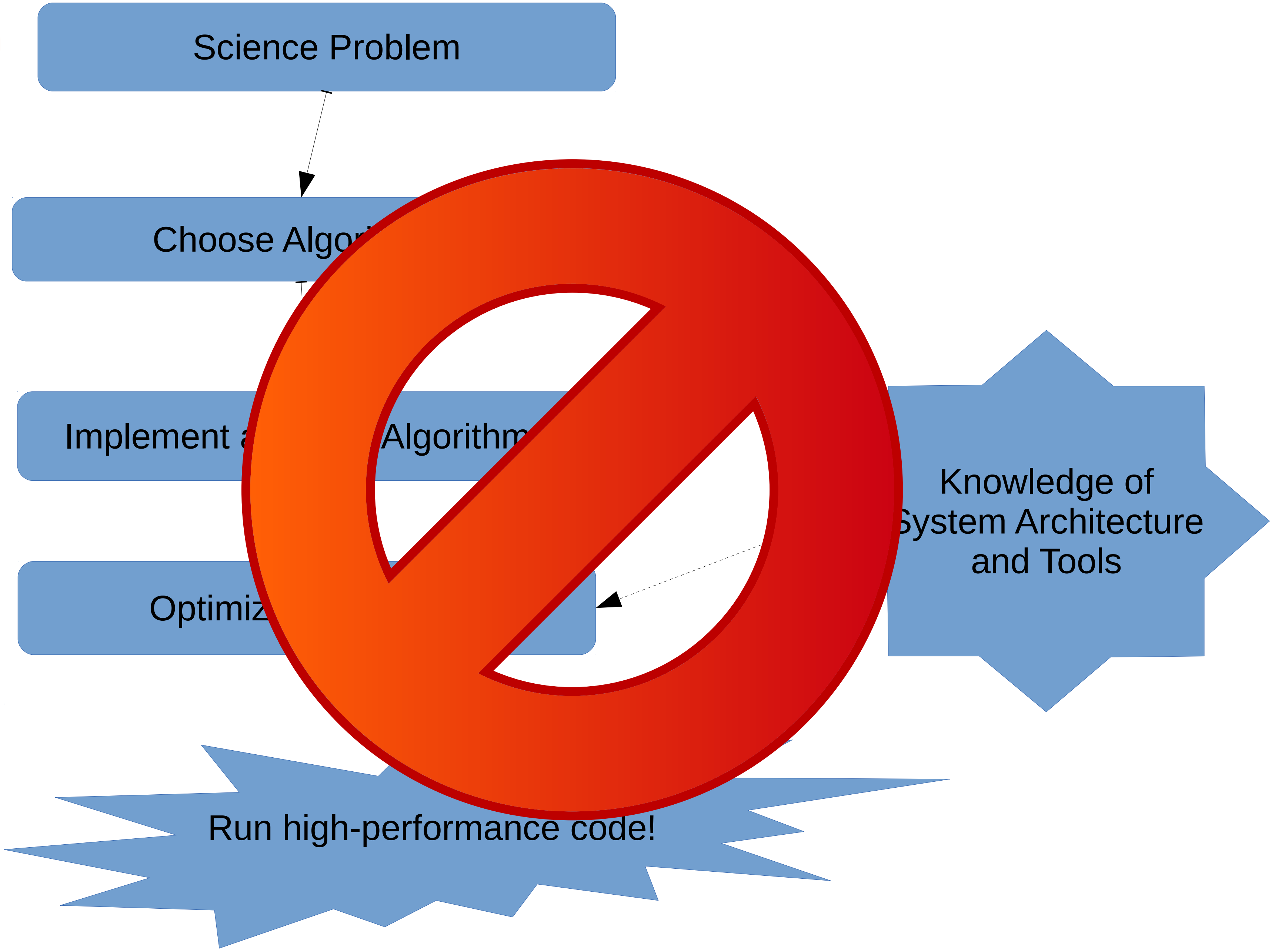


- Some trends in HPC architectures
- How you can optimize your code for these architectures
- Q&A

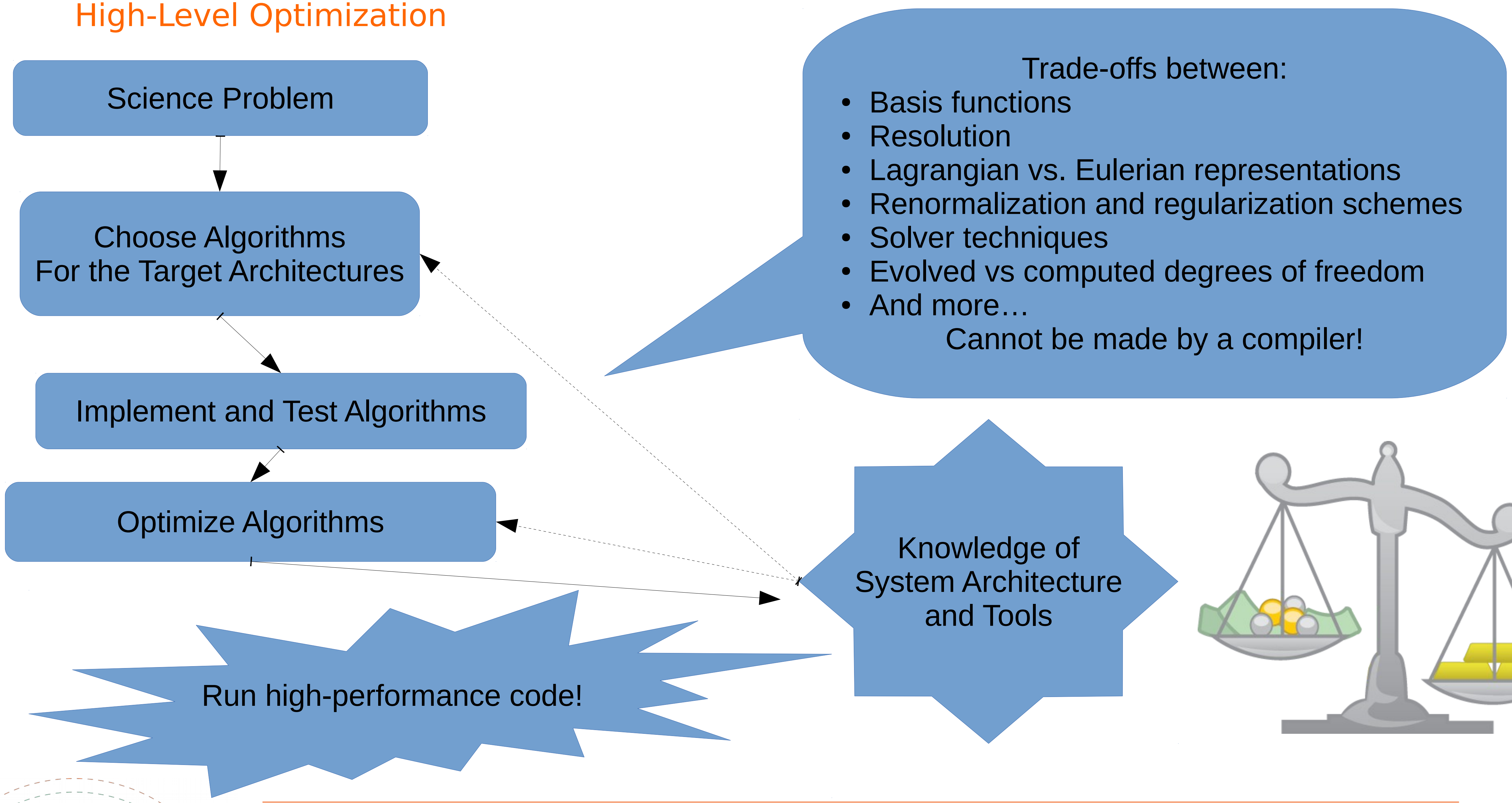
High-Level Optimization


Science Problem
Choose Algorithms
Implement and Test Algorithms
Optimize Algorithms
Knowledge of System Architecture and Tools
Run high-performance code!

High-Level Optimization


Science Problem
Choose Algor
Implement a Algorithm
Optimiz
Knowledge of System Architecture and Tools
Run high-performance code!

High-Level Optimization


Science Problem
Choose Algorithms For the Target Architectures
Implement and Test Algorithms
Optimize Algorithms
Run high-performance code!
Knowledge of System Architecture and Tools

Trade-offs between:

- Basis functions
- Resolution
- Lagrangian vs. Eulerian representations
- Renormalization and regularization schemes
- Solver techniques
- Evolved vs computed degrees of freedom
- And more…

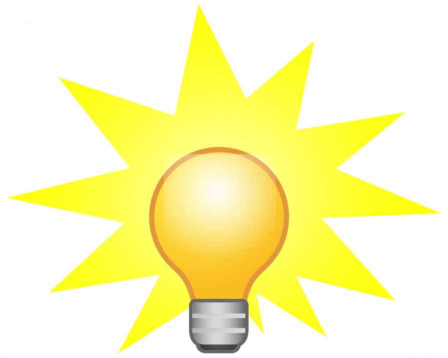
Cannot be made by a compiler!

Computer Architecture

Traditional computers are built to:

- Move data
- Make decisions
- Compute polynomials (of relatively-low order)

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$



Computer Architecture

```
$ cat /tmp/f0.c
#include <math.h>

double foo(double a0, double a1, double a2, double a3, double a4, double x) {
  return a0 + a1*x + a2*pow(x, 2) + a3*pow(x, 3) + a4*pow(x, 4);
}

$ gcc -O3 -S -o - /tmp/f0.c
…
        movsd   %xmm0, 8(%rsp)
        movapd  %xmm5, %xmm0
        movsd   %xmm1, 56(%rsp)
        movsd   .LC0(%rip), %xmm1
        movsd   %xmm2, 48(%rsp)
        movsd   %xmm3, 40(%rsp)
        movsd   %xmm4, 32(%rsp)
        movsd   %xmm5, 24(%rsp)
        call  pow
…
        call  pow
…
```

Not useful work.

These calls are expensive!

Yes, -ffast-math will fix this...

Computer Architecture

```
$ cat /tmp/f.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
  return a0 + a1*x + a2*x*x + a3*x*x*x + a4*x*x*x*x;
}
```

```
$ gcc -O3 -S -o - /tmp/f.c
…
        mulsd   %xmm5, %xmm1
        mulsd   %xmm5, %xmm2
        mulsd   %xmm5, %xmm4
        mulsd   %xmm5, %xmm3
        addsd   %xmm1, %xmm0
        mulsd   %xmm5, %xmm2
        mulsd   %xmm5, %xmm4
        mulsd   %xmm5, %xmm3
        addsd   %xmm2, %xmm0
        mulsd   %xmm5, %xmm3
        movapd  %xmm0, %xmm2
        movapd  %xmm4, %xmm0
        addsd   %xmm3, %xmm2
        mulsd   %xmm5, %xmm0
        mulsd   %xmm0, %xmm5
        addsd   %xmm5, %xmm2
        movapd  %xmm2, %xmm0
        ret
```

This is better, but...

Computer Architecture

```
$ cat /tmp/f1.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
  return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}
```

```
$ gcc -O3 -S -o - /tmp/f1.c
…
        mulsd     %xmm5, %xmm4
        addsd     %xmm4, %xmm3
        mulsd     %xmm5, %xmm3
        addsd     %xmm3, %xmm2
        mulsd     %xmm5, %xmm2
        addsd     %xmm2, %xmm1
        mulsd     %xmm5, %xmm1
        addsd     %xmm1, %xmm0
        ret
```

And this is better, but...

```
$ cat /tmp/f1.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
  return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}
```

PowerPC, etc. uses -mcpu= instead of -march=

And remember the correct target flags...

```
$ gcc -O3 -S -o - /tmp/f1.c
…
        mulsd   %xmm5, %xmm4
        addsd   %xmm4, %xmm3
        mulsd   %xmm5, %xmm3
        addsd   %xmm3, %xmm2
        mulsd   %xmm5, %xmm2
        addsd   %xmm2, %xmm1
        mulsd   %xmm5, %xmm1
        addsd   %xmm1, %xmm0
        ret
```

```
$ gcc -O3 -S -o - /tmp/f1.c -march=native
…
        vfmadd231sd %xmm5, %xmm4, %xmm3
        vfmadd231sd %xmm3, %xmm5, %xmm2
        vfmadd231sd %xmm2, %xmm5, %xmm1
        vfmadd231sd %xmm1, %xmm5, %xmm0
        ret
```

A fused multiply-add

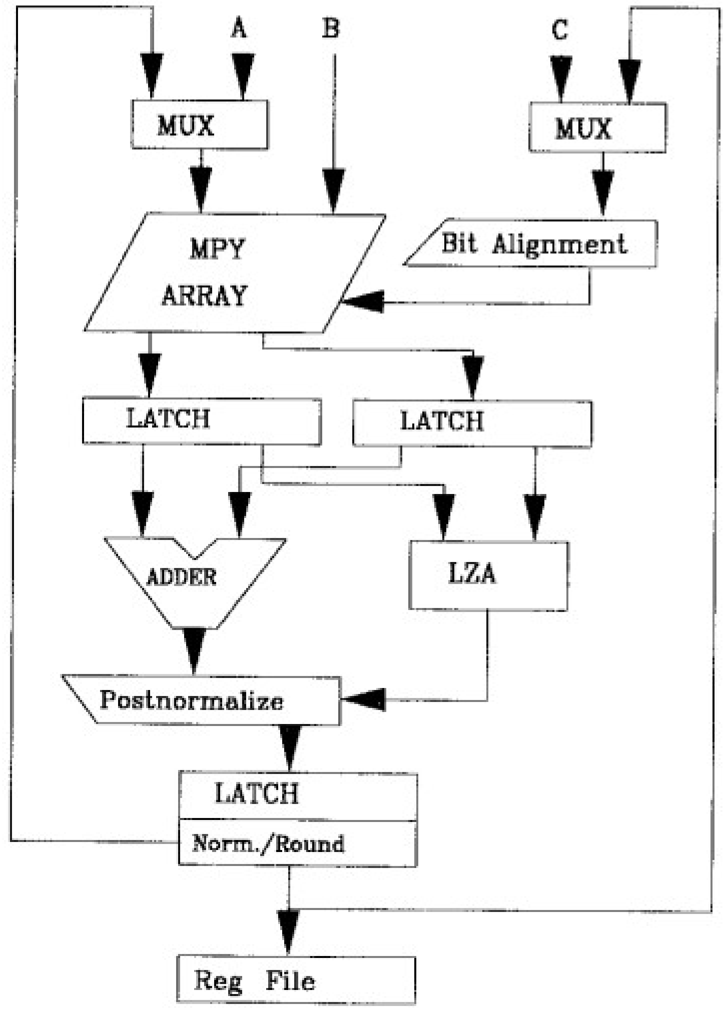
Computer Architecture

```
$ cat /tmp/f1.c
double foo(double a0, ..., double x) {
  return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}
```

```
t0 = fma(a4, x, a3)
t1 = fma(t0, x, a2)
t2 = fma(t1, x, a1)
t3 = fma(t2, x, a0)
return t3
```

But floating-point is complicated, so each operation cannot be completed in one clock cycle. ~6 clock cycles are needed.

IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 25, NO. 5, OCTOBER 1990



Computer Architecture

But this is not good…

```
t0 = fma(a4, x, a3)
Waiting…
Waiting…
Waiting…
Waiting…
Waiting...
t1 = fma(t0, x, a2)
...
t2 = fma(t1, x, a1)
...
t3 = fma(t2, x, a0)
…
return t3
```

A lot of computer architecture revolves around this question:

How do we put useful work here?

Hardware Threads

One way is to use hardware threads...

```
t0 = fma(a4, x, a3) [thread 0]
t0 = fma(a4, x, a3) [thread 1]
t0 = fma(a4, x, a3) [thread 2]
t0 = fma(a4, x, a3) [thread 3]
t0 = fma(a4, x, a3) [thread 4]
t0 = fma(a4, x, a3) [thread 5]
t1 = fma(t0, x, a2)
...
t2 = fma(t1, x, a1)
...
t3 = fma(t2, x, a0)
…
return t3
```

These can be OpenMP threads, pthreads, or, on a CPU, different processes.

How many threads do we need?
How much latency do we need to hide?

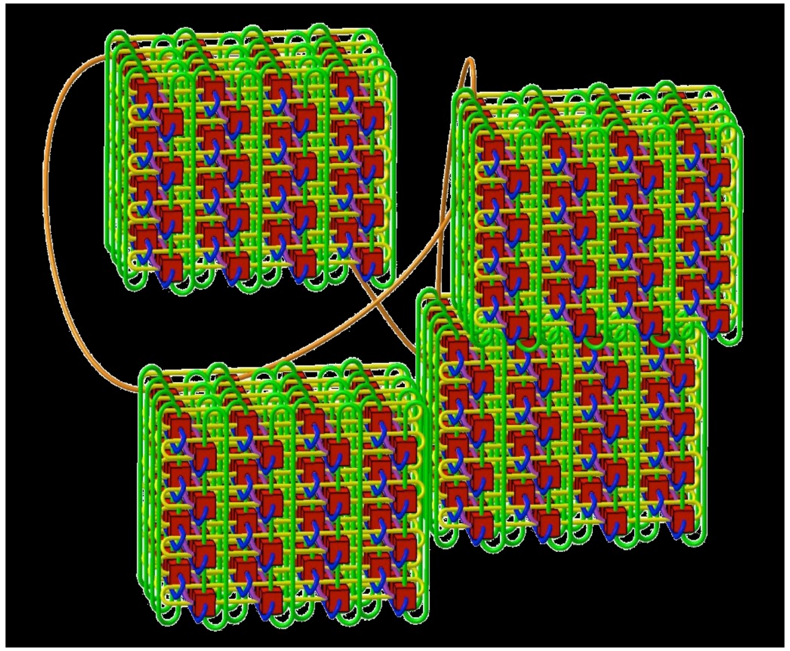
Time Scales in Computing

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD
Disk seek                           10,000,000   ns   10,000 us   10 ms
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms   80x memory
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

Latency Numbers Every Programmer Should Know: https://gist.github.com/jboner/2841832

The IBM BG/Q network is fast...

- Each A/B/C/D/E link bandwidth: 4 GB/s
- Bisection bandwidth (32 racks): 13.1 TB/s
- HW latency
  - Best: 80 ns (nearest neighbor)
  - Worst: 3 µs (96-rack 20 PF system, 31 hops)
- MPI latency (zero-length, nearest-neighbor): 2.2 µs

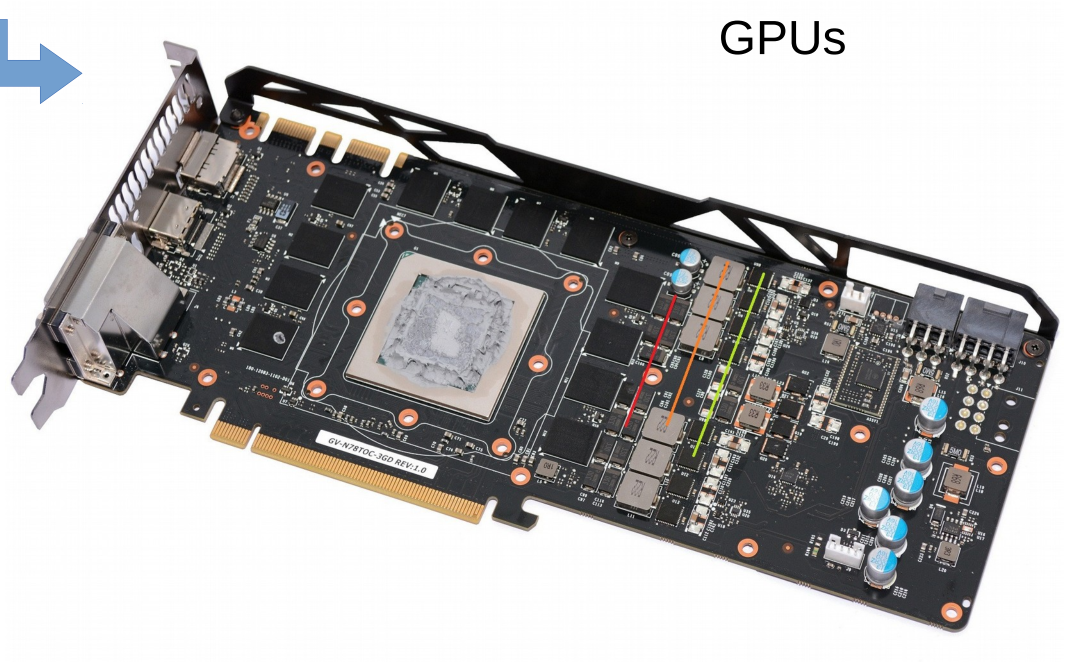


MPI does add overhead
which is generally minimal.
If you're sensitive to it, you can
use PAMI (or the SPI interface) directly

Supercomputing “Swim Lanes”

“Many Core” CPUs

GPUs



https://forum.beyond3d.com/threads/nvidia-pascal-speculation-thread.55552/page-4

http://www.nextplatform.com/2015/11/30/inside-future-knights-landing-xeon-phi-systems/

Supercomputing “Swim Lanes”

“Many Core” CPUs

GPUs

- 4 hardware threads per core
- To make up the rest, relies on:
  - OOO processing with branch prediction
  - Loop unrolling
  - SIMD (vectorization)

- Lots of hardware threads
- Many hardware threads share the instruction stream (SIMT)

Many threads, but SIMT minimizes the per-thread control state/logic.

Some CUDA Terminology

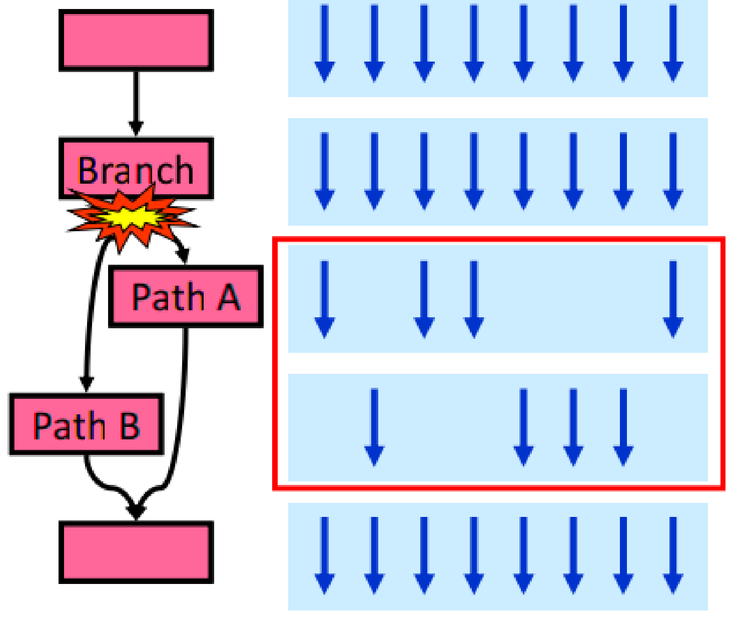
- Dispatched threads are organized into a grid; all threads in a grid execute the same kernel function
- A grid is decomposed into a 2D array of blocks (gridDim.x by gridDim.y)
- Each block is decomposed into a 3D array of threads (blockDim.x by blockDim.y by blockDim.z)
- The size of each block is limited to 1024 threads
- Threads in different blocks cannot synchronize (using __syncthreads() - they might execute in any order)

```
__global__ void add(int *a, int *b, int *c) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  c[index] = a[index] + b[index];
}
```

Each thread has access to its coordinates and grid/block dimensions so it can figure out what to do.

SIMT

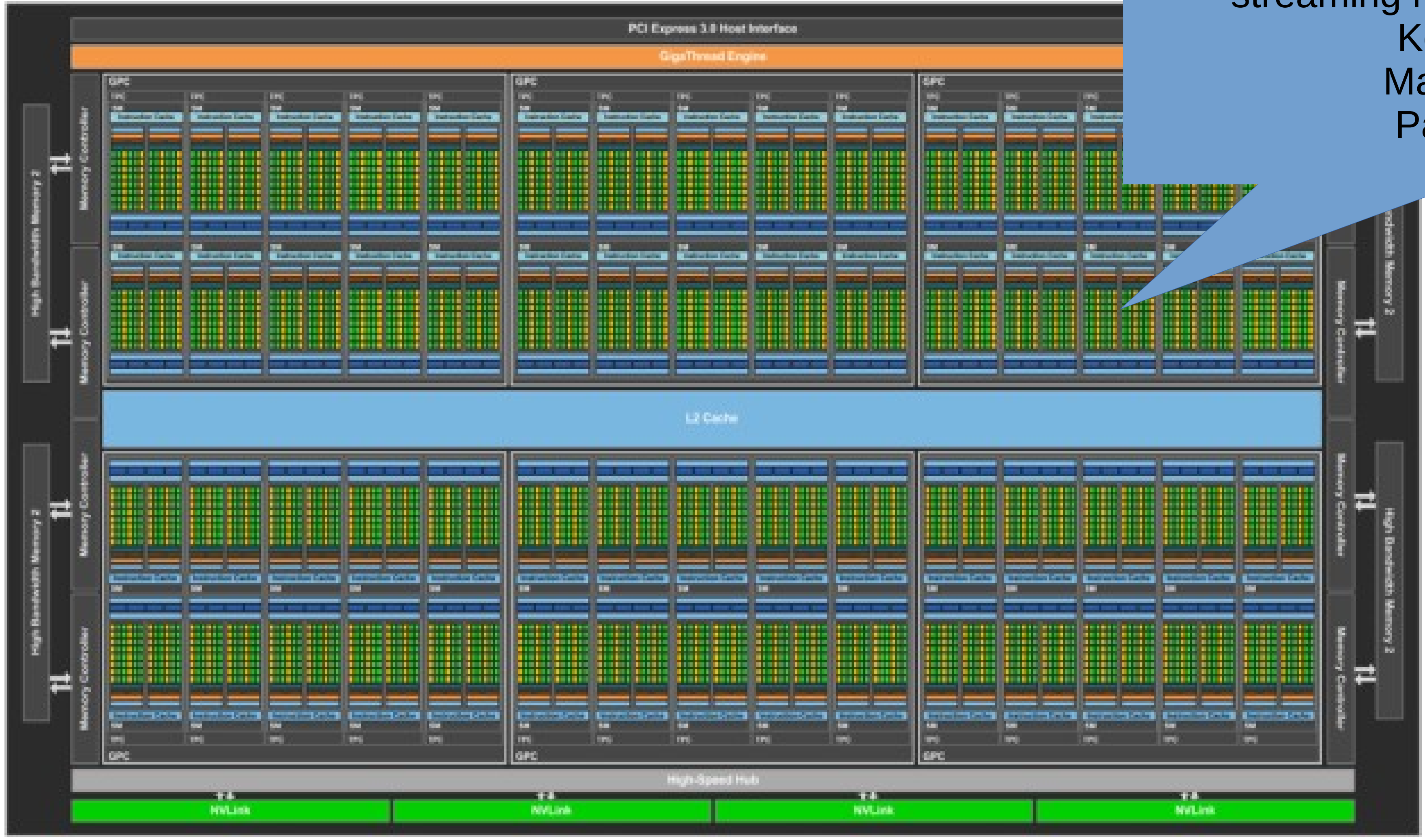
- Threads are divided into groups of 32, called “warps” in NVIDIA's terminology (AMD calls these “wavefronts” - with a size of 64)
- All threads in each warp share many instruction stream resources (i.e. they have the same instruction pointer)
- The size of a warp is akin to the number of vector lanes on a CPU's SIMD unit
- Beware of branch divergence...



Pretend there are 32 arrows per line

http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf

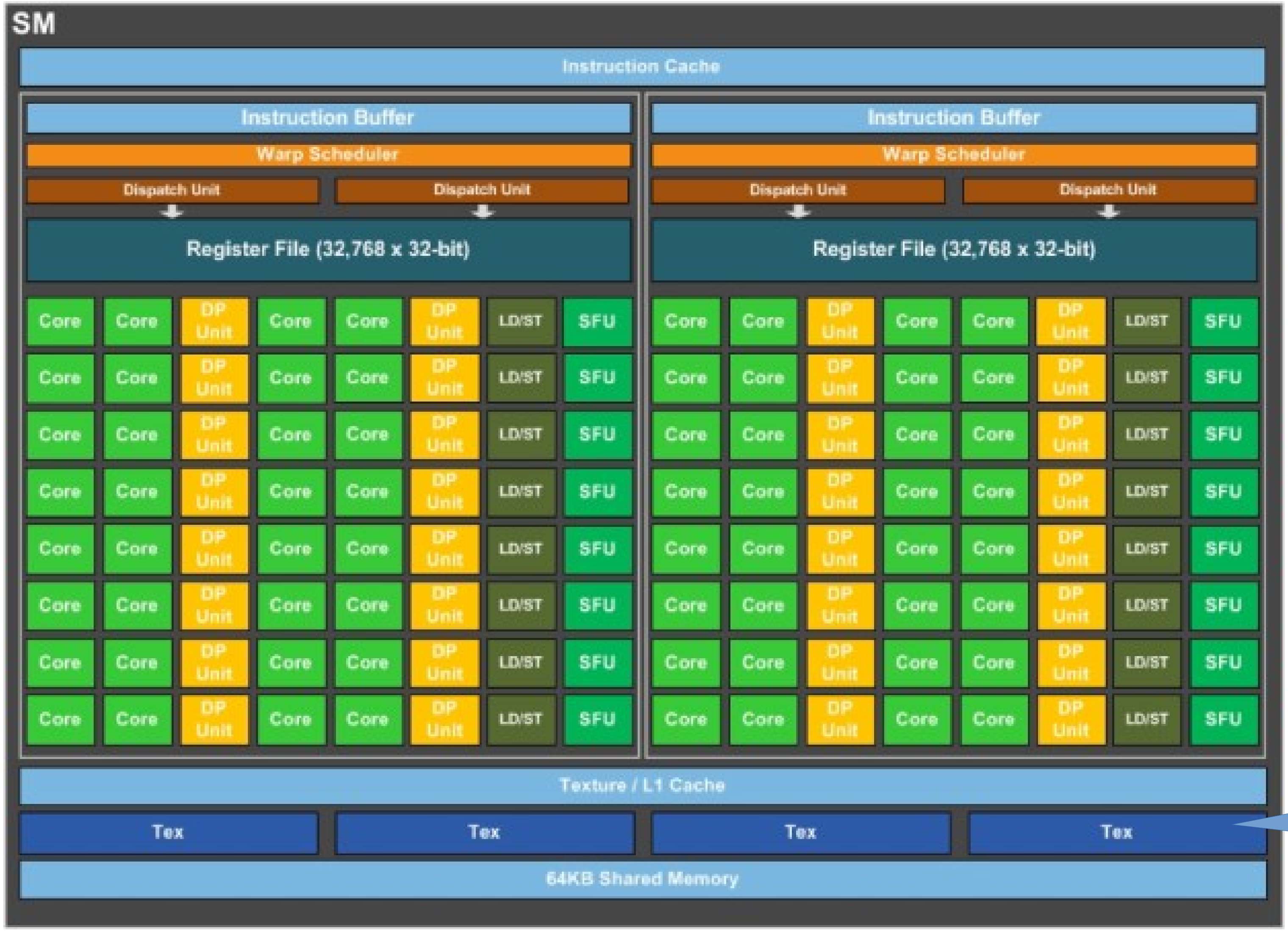
GPU Layout



GPUs have “cores”, but NVIDIA calls them “streaming multiprocessors” or SMs:
Kepler: 15 SMs
Maxwell: 24 SMs
Pascal: 56 SMs

https://devblogs.nvidia.com/parallelforall/inside-pascal/

GPU SM



- Single precision / SM
  - Pascal: 64
  - Kepler: 192
- Double precision / SM
  - Pascal: 32
  - Kepler: 64
- Max 64 warps / SM
- Max blocks / SM
  - Pascal: 32
  - Kepler: 16

Put “read only” data here

https://devblogs.nvidia.com/parallelforall/inside-pascal/

Register Pressure!

- Each SM has a 256 KB register file
- Each thread can use up to 255 32-bit registers
- An SM running its maximum 2048 threads, however, could support only ~32 registers / thread!

```
t0 = fma(a4, x, a3)
t1 = fma(t0, x, a2)
t2 = fma(t1, x, a1)
t3 = fma(t2, x, a0)
return t3
```

This calculation needs ~3 registers:
one for x, one for a<n>, one for t<n>
But the compiler might use more by default!
(see docs on __launch_bounds__)

Unlike on a CPU, after the first 32, there is a significant cost to the incremental use of each register!

Loop Unrolling

CPUs have a fixed register file per thread, and the compiler can use that to hide latency...

```
for (int i = 0; i < n; ++i) {
  x = Input[i]
  t0 = fma(a4, x, a3)
  t1 = fma(t0, x, a2)
  t2 = fma(t1, x, a1)
  t3 = fma(t2, x, a0)
  Output[i] = t3
}
```

unroll by 2

```
for (int i = 0; i < n; i += 2) {
  x = Input[i]
  y = Input[i+1]
  t0 = fma(a4, x, a3)
  u0 = fma(a4, y, a3)
  t1 = fma(t0, x, a2)
  u1 = fma(u0, y, a2)
  t2 = fma(t1, x, a1)
  u2 = fma(u1, y, a1)
  t3 = fma(t2, x, a0)
  u3 = fma(u2, y, a0)
  Output[i] = t3
  Output[i+1] = u3
}
```

I hope these are in cache

Each pair is independent, so no waiting in between dispatches

Showing unroll by 2 so it fits on the slide, you need to unroll by more to fully hide FP or L1 latency

If you need to tune this yourself, most compilers have a '#pragma unroll' feature.

CPU Registers

You can't unroll enough to completely hide anything but “on core” latencies (e.g. L1 cache hits and from FP pipeline) – you just don't have enough registers!

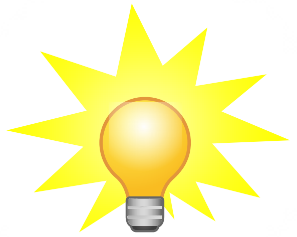
- x86_64 has 16 general-purpose registers (GPRs) – for scalar integer data, pointers, etc. – and 16 floating-point/vector registers
- With AVX-512 (e.g. with Knights Landing) there are 32 floating-point/vector registers
- AVX-512 also adds 8 operation mask registers
- PowerPC has 32 GPRs, 32 scalar floating-point registers and 32 vector registers (modern cores with VSX effectively combine these into 64 floating-point/vector registers)

OOO Execution and Loops

- CPUs, including Intel's Knights Landing, use out-of-order (OOO) execution to hide latency
- So to say that there are only 16 GPRs, for example, isn't the whole story: there are just 16 GPRs that the compiler can name

```
for (int i = 0; i < n; ++i) {
  x = Input[i]
  t0 = fma(a4, x, a3)
  t1 = fma(t0, x, a2)
  t2 = fma(t1, x, a1)
  t3 = fma(t2, x, a0)
  Output[i] = t3
}
```

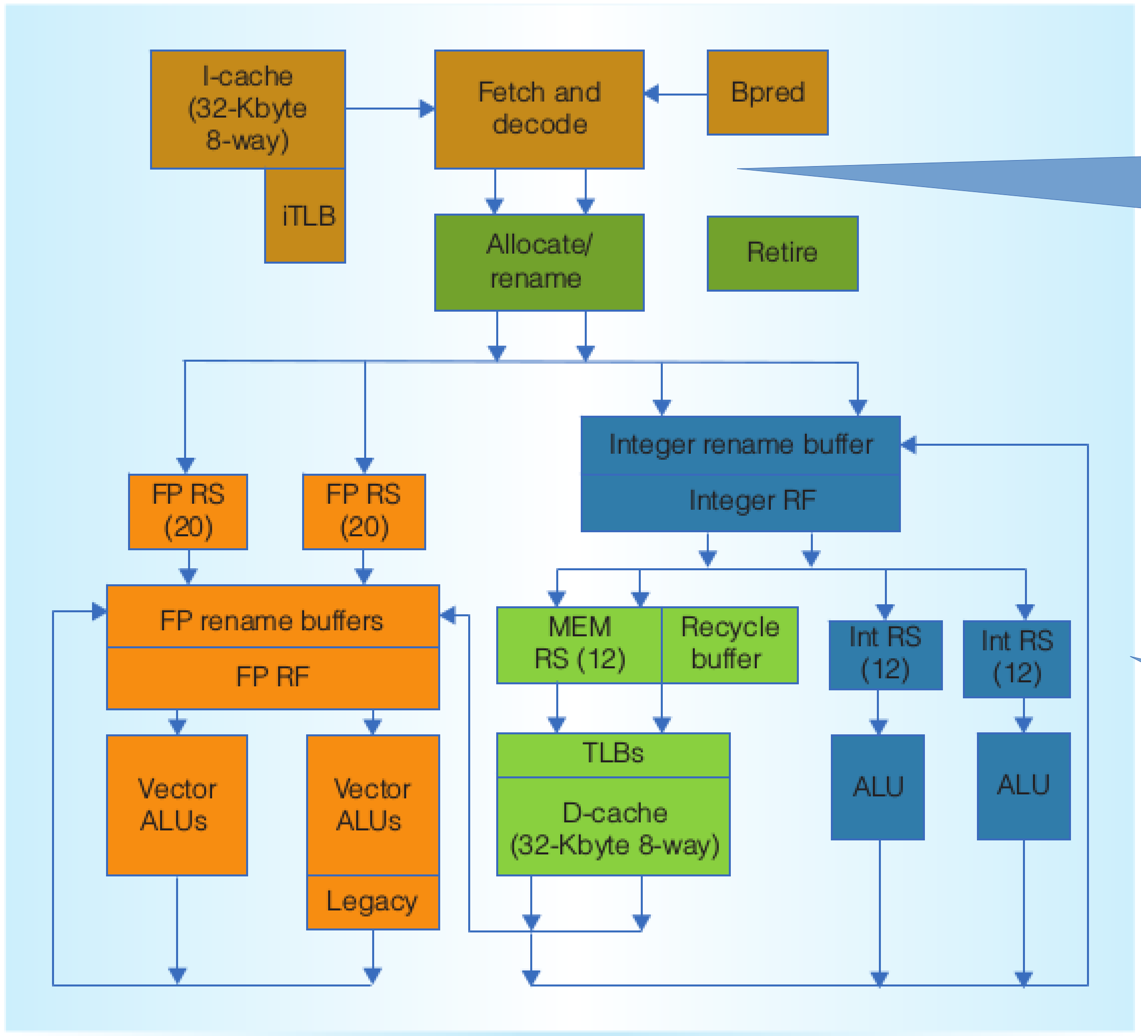
Processor can predict this will be true, and can start issuing instructions for multiple iterations at a time!



OOO Execution

- Importing to exploiting instruction-level parallelism (ILP) – each core's multiple pipelines
- Combined with branch prediction, can effectively provide a kind of dynamic loop unrolling
- Limited by the number of “rename buffer entries” (72 on Knights Landing)
- Limited by the number of “reorder buffer entries” (72 on Knights Landing)
- Mispredicted branches can lead to wasted work!

KNL Pipeline



Fetch/decode 16 bytes per cycle (i.e. two instructions per cycle)
Careful: AVX-512 instructions can be up to 12 bytes each if they have non-compressed displacements!

2 FP/vector operations,
2 memory operations,
and 2 scalar integer operations per cycle!

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7453080

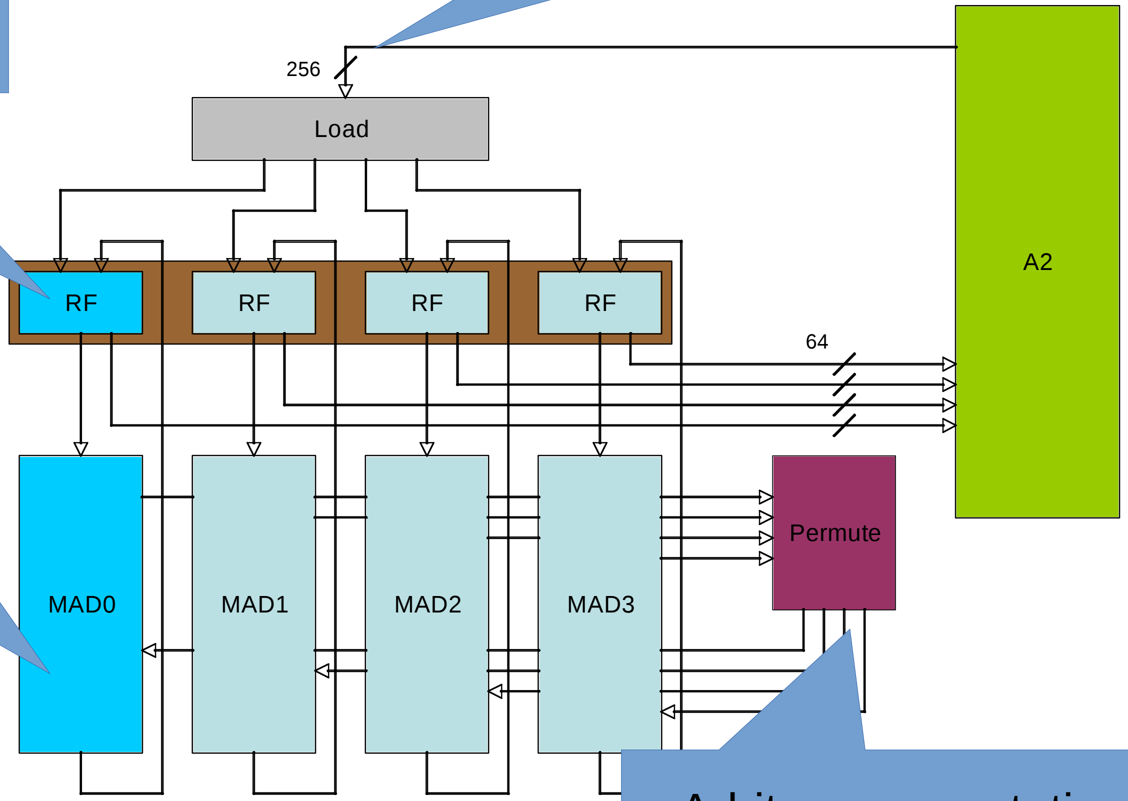
Vectorization: The Quad-Processing eXtension (QPX)

32 QPX registers (and 32 general purpose registers) per thread

The first vector element in each vector register is the corresponding scalar FP register.

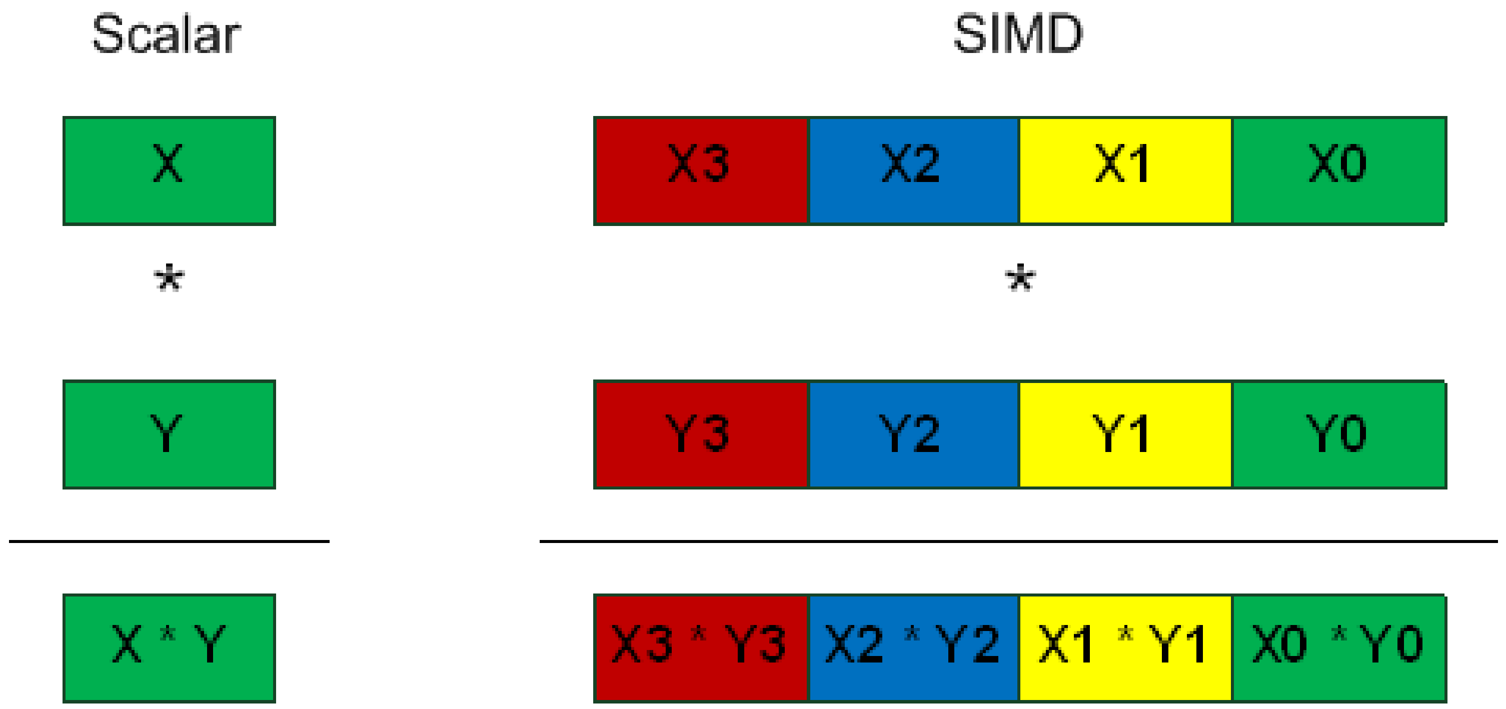
FP arithmetic completes in six cycles (and is fully pipelined). Loads/stores execute in the XU pipeline (same as all other load/stores).

Arbitrary permutations complete in only two cycles.



(This is for the IBM BG/Q, but the picture is fairly generic)

SIMD: What does it mean?

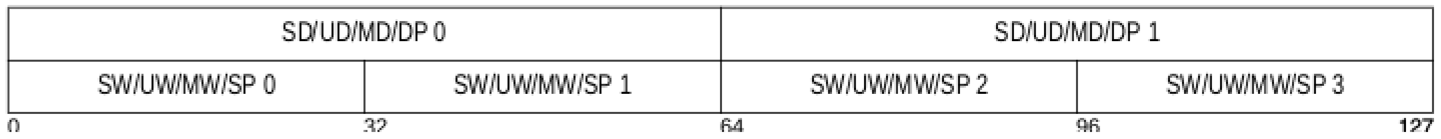


https://software.intel.com/en-us/articles/ticker-tape-part-2

Autovectorization (or manual vectorization)

Vectors Have Many Types

- A 512-bit vector can hold 8 double-precision numbers, 16 single-precision numbers, etc.
- Different assembly instructions have different assumptions about the data types
- Except on the IBM BG/Q (where only FP is supported), both integer and FP types are supported

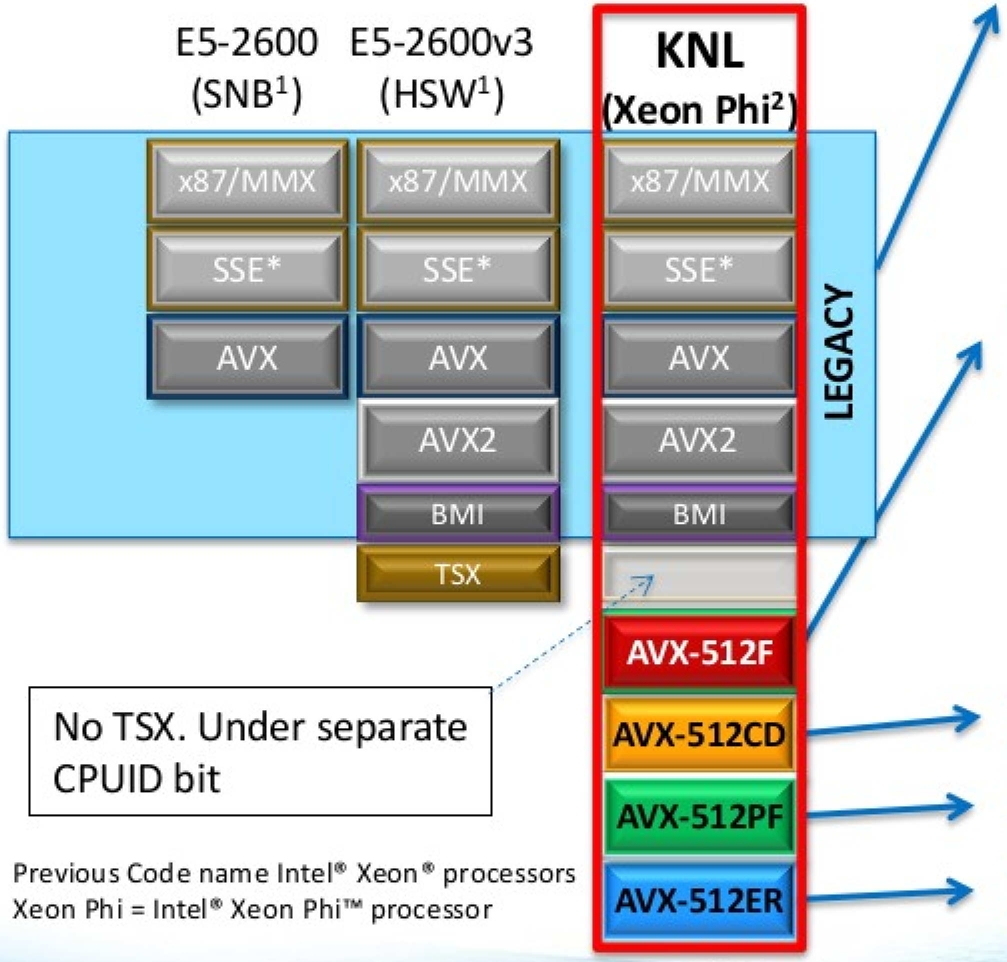


The same vector register can be divided in different ways

(This diagram is from the IBM POWER ISA manual, showing the 128-bit VSX registers)

		Vector Length		
		128	256	512
	Byte	16	32	64
	Word	8	16	32
element	Dw ord/SP	4	8	16
size	Qw ord/DP	2	4	8

KNL ISA



KNL implements all legacy instructions

- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions

- 512-bit FP/Integer Vectors
- 32 registers, & 8 mask registers
- Gather/Scatter

Conflict Detection: Improves Vectorization

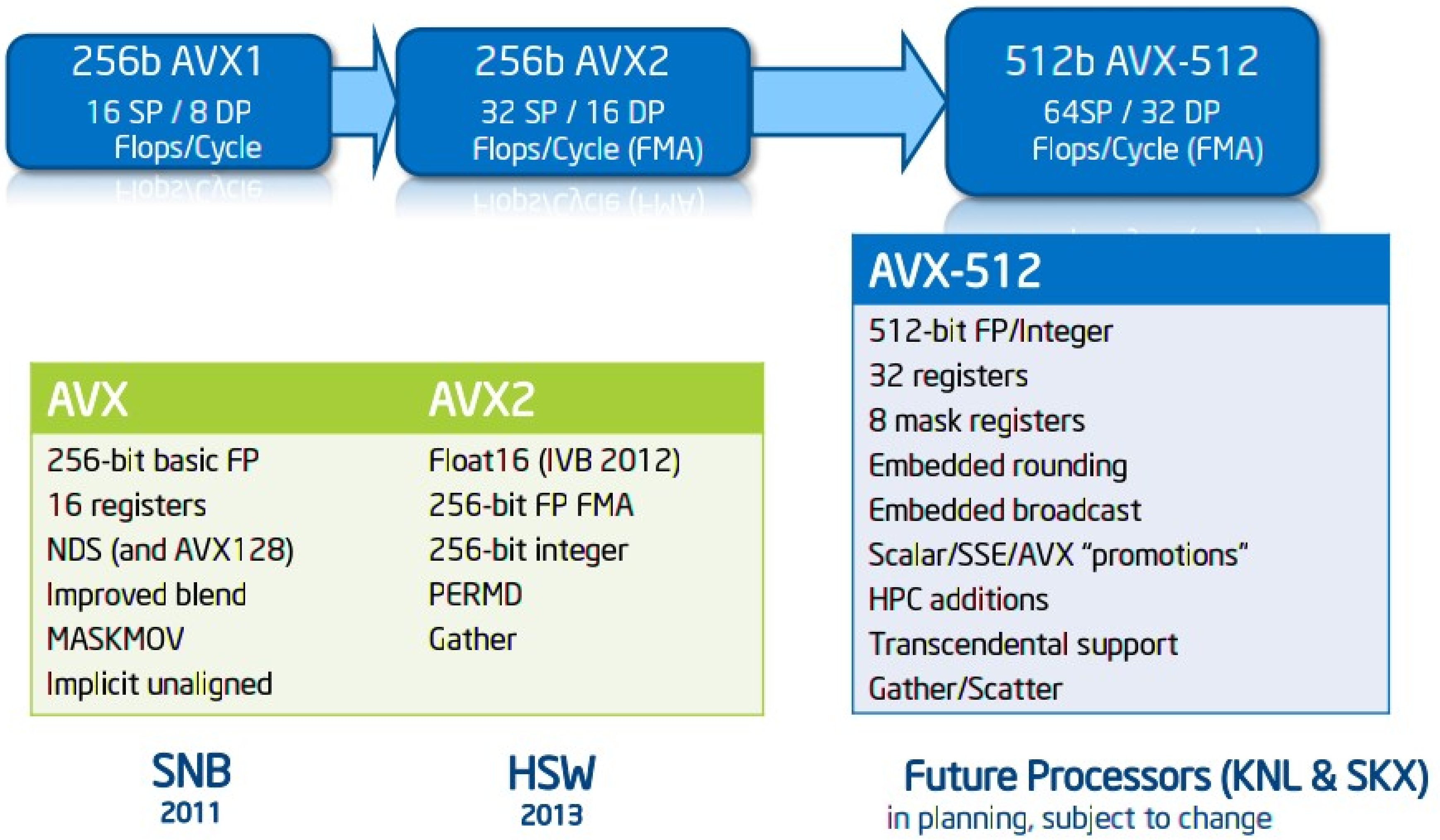
Prefetch: Gather and Scatter Prefetch

Exponential and Reciprocal Instructions

1. Previous Code name Intel® Xeon® processors
2. Xeon Phi = Intel® Xeon Phi™ processor

Avinash Sodani CGO PPoPP HPCA Keynote 2016

What's in AVX-512?



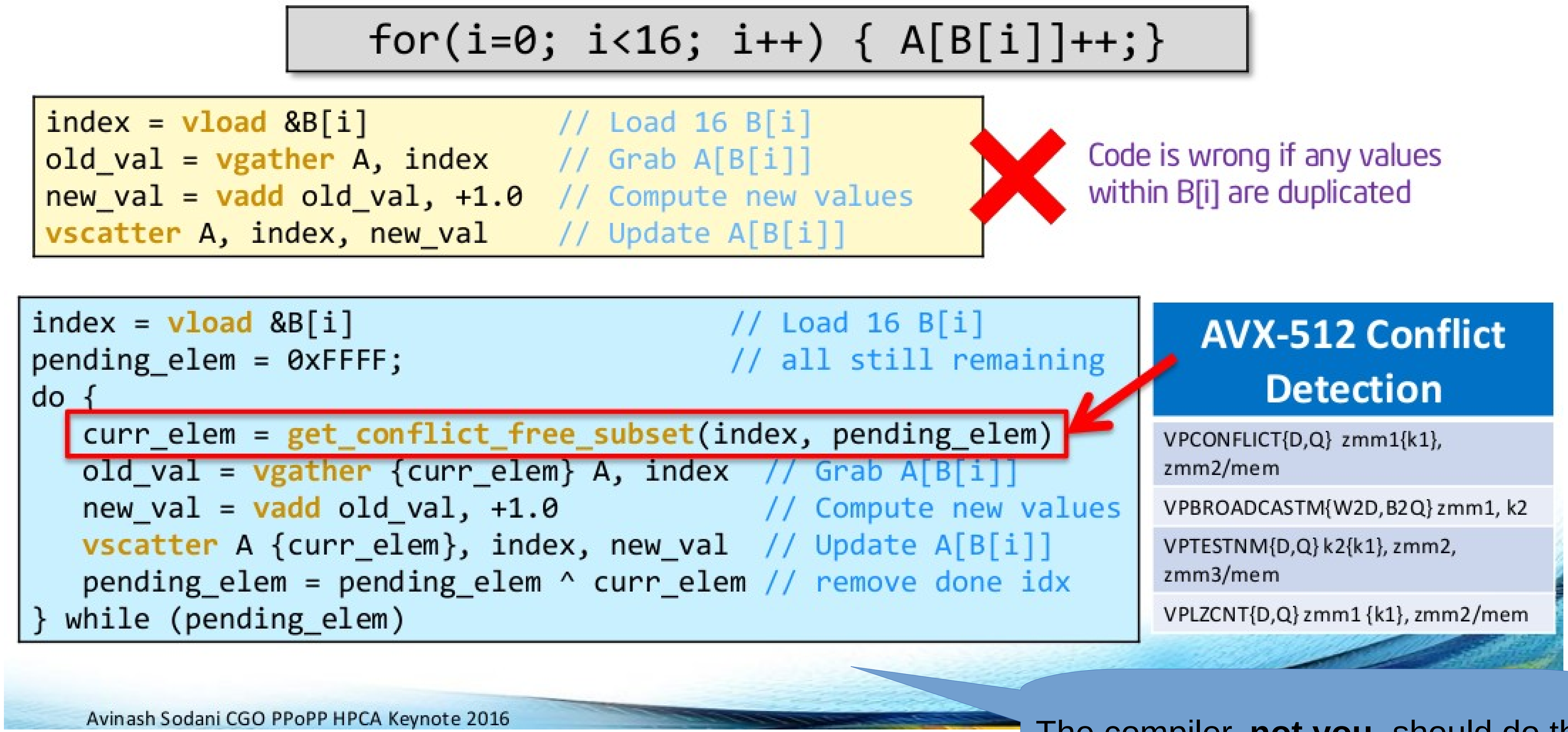
256b AVX1
16 SP / 8 DP Flops/Cycle
256b AVX2
32 SP / 16 DP Flops/Cycle (FMA)
512b AVX-512
64SP / 32 DP Flops/Cycle (FMA)
SNB
2011
HSW
2013
Future Processors (KNL & SKX)
in planning, subject to change

AVX	AVX2
256-bit basic FP	Float16 (IVB 2012)
16 registers	256-bit FP FMA
NDS (and AVX128)	256-bit integer
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

AVX-512
512-bit FP/Integer
32 registers
8 mask registers
Embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
HPC additions
Transcendental support
Gather/Scatter

https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

KNL AVX512-CD



```
for(i=0; i<16; i++) { A[B[i]]++;}
```

```
index = vload &B[i]            // Load 16 B[i]
old_val = vgather A, index     // Grab A[B[i]]
new_val = vadd old_val, +1.0   // Compute new values
vscatter A, index, new_val     // Update A[B[i]]
```

Code is wrong if any values within B[i] are duplicated

```
index = vload &B[i]                             // Load 16 B[i]
pending_elem = 0xFFFF;                          // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index    // Grab A[B[i]]
   new_val = vadd old_val, +1.0              // Compute new values
   vscatter A {curr_elem}, index, new_val    // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem   // remove done idx
} while (pending_elem)
```

AVX-512 Conflict Detection

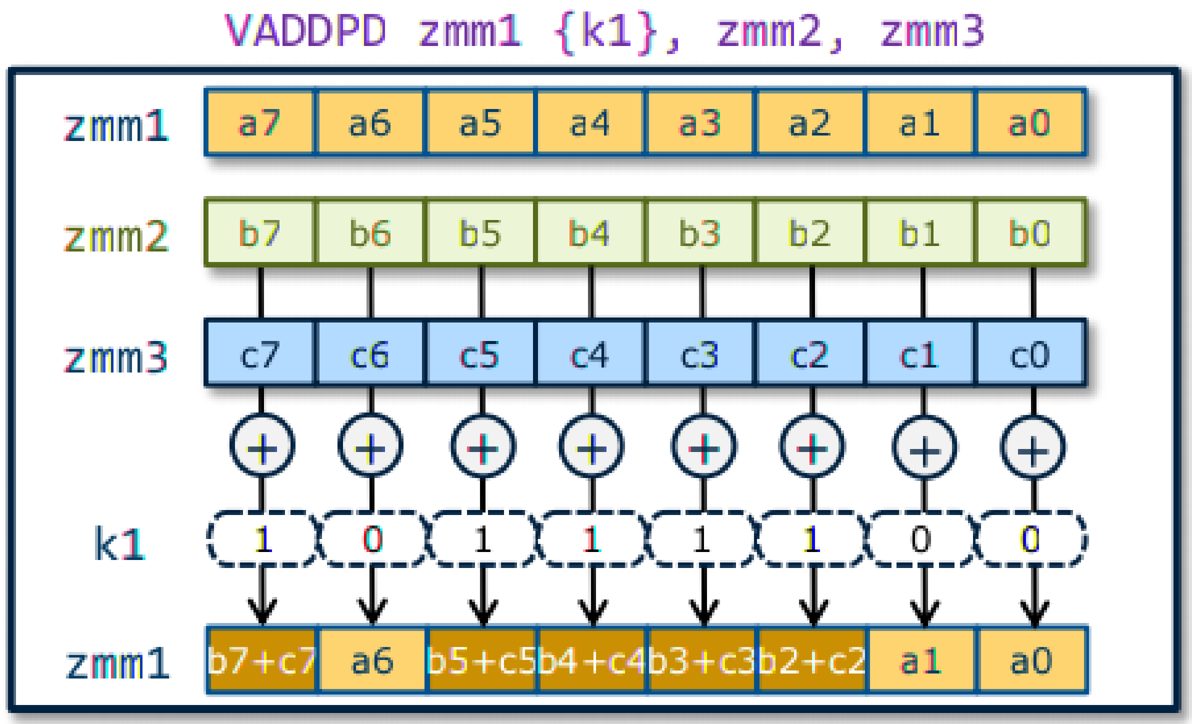
VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
VPBROADCASTM{W2D,B2Q} zmm1, k2
VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

Avinash Sodani CGO PPoPP HPCA Keynote 2016

The compiler, **not you**, should do this!

Argonne Leadership Computing Facility

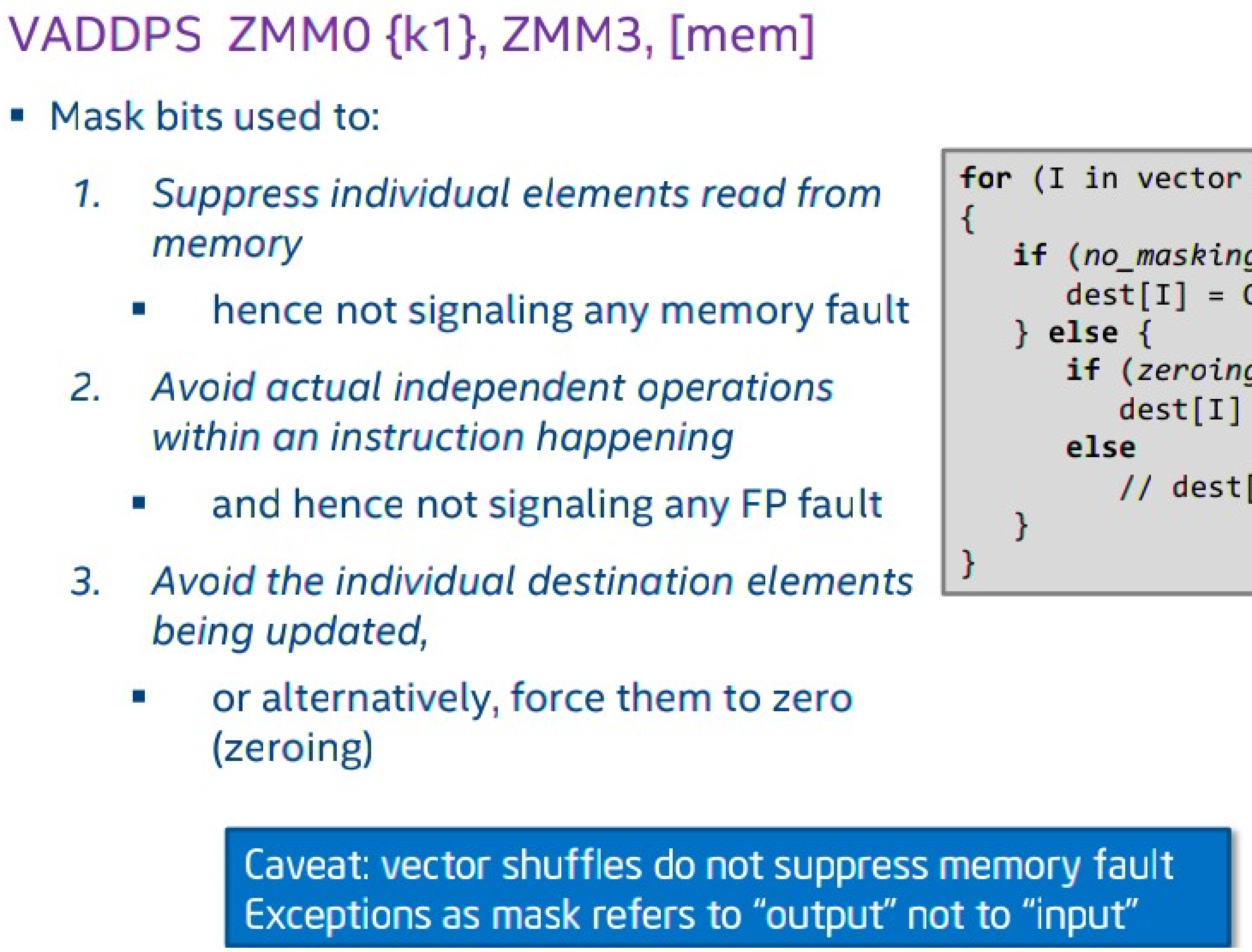
AVX-512 Mask Registers



https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

AVX-512 has 8 mask registers (64-bits each)

Why Masking is Special…



VADDPS ZMM0 {k1}, ZMM3, [mem]

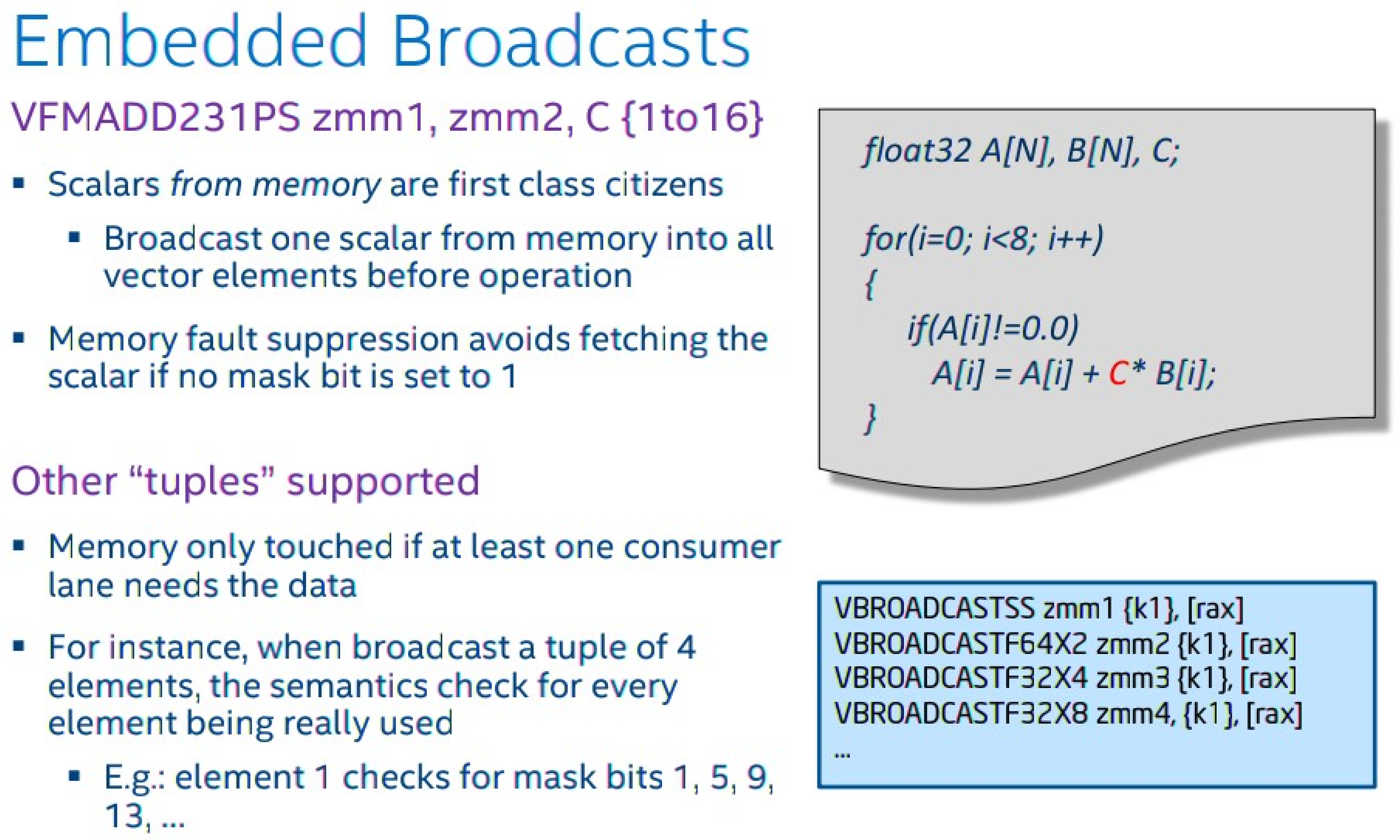
- Mask bits used to:
  1. *Suppress individual elements read from memory*
     - hence not signaling any memory fault
  2. *Avoid actual independent operations within an instruction happening*
     - and hence not signaling any FP fault
  3. *Avoid the individual destination elements being updated,*
     - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
   if (no_masking or mask[I]) {
      dest[I] = OP(src2, src3)
   } else {
      if (zeroing_masking)
         dest[I]  = 0
      else
         // dest[I] is preserved
   }
}
```

Caveat: vector shuffles do not suppress memory fault Exceptions as mask refers to “output” not to “input”

https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

AVX-512 Embedded Broadcasts



Embedded Broadcasts

VFMADD231PS zmm1, zmm2, C {1to16}

- Scalars *from memory* are first class citizens
  - Broadcast one scalar from memory into all vector elements before operation
- Memory fault suppression avoids fetching the scalar if no mask bit is set to 1

```
float32 A[N], B[N], C;

for(i=0; i<8; i++)
{
    if(A[i]!=0.0)
      A[i] = A[i] + C* B[i];
}
```

Other "tuples" supported

- Memory only touched if at least one consumer lane needs the data
- For instance, when broadcast a tuple of 4 elements, the semantics check for every element being really used
  - E.g.: element 1 checks for mask bits 1, 5, 9, 13, ...

```
VBROADCASTSS zmm1 {k1}, [rax]
VBROADCASTF64X2 zmm2 {k1}, [rax]
VBROADCASTF32X4 zmm3 {k1}, [rax]
VBROADCASTF32X8 zmm4, {k1}, [rax]
...
```

https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf

Why Masking Matters?

```
void foo(float * restrict x, float * restrict y, float * restrict z, float * restrict v, float * restrict out, int n) {
  for (int i = 0; i < n; ++i) {
    float r2 = x[i]*x[i] + y[i]*y[i] + z[i]*z[i];
    if (r2 > eps) {
      out[i] = f(v[i], r2);
    } else {
      out[i] = 0;
    }
  }
}
```

Traditionally, a compiler could not autovectorize this!
(not a pointer aliasing problem)

To vectorize, we essentially convert this into (m == # vector lanes):
r2[i:i+m] = <r2_i, r2_{i+1}, ...>
out[i:i+m] = r2[i:i+m] > <eps, eps, …> ? f(v[i:i+m], r2[i:i+m]) : <0, 0, … >

Why? The compiler needs to deal with this (hypothetical) situation:

What if it were the case that the array “v” was not as long as x, y, and z (i.e. < n), but the programmer has arranged that (r2 > eps) will be false for all indices i invalid for the array v?

With AVX-512 masking, this is not a problem (we can mask off the access we don't need).

Note: Fortran (potentially) does not have this problem, even without masking (it knows the length of the arrays)!

Digression on “restrict”

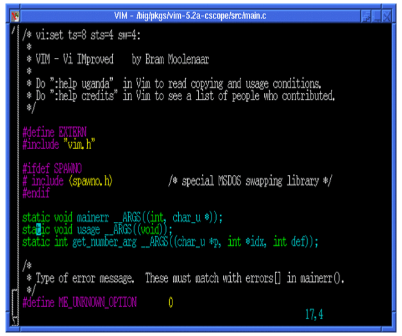
```
void foo(float * restrict x, float * restrict y, float * restrict z, float * restrict v, float * restrict out, int n) {
  ...
}
```

“restrict” only a keyword in C. Use __restrict in C++

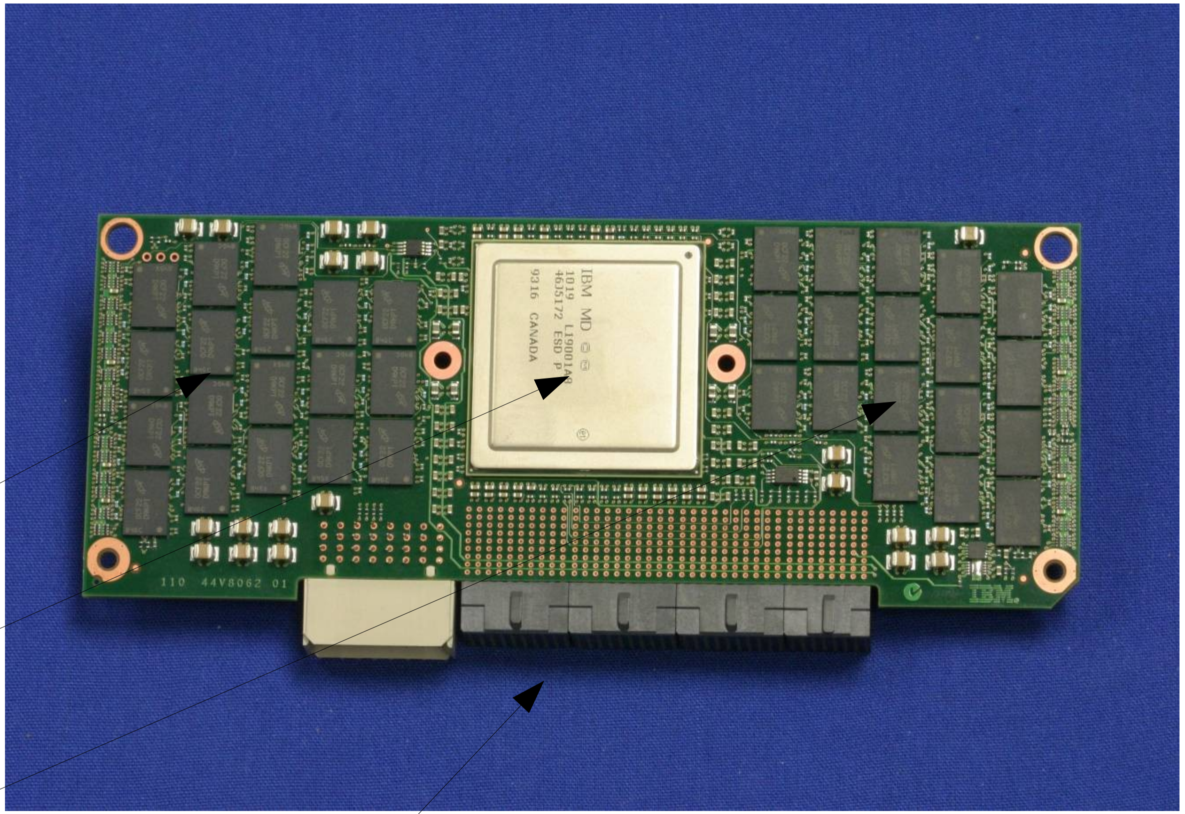
Not that restrict goes on the variable,
not the pointer type!
Not: restrict float *out

restrict means: Within the scope of the restrict-qualified variable, memory accessed through that pointer, or any pointer based on it, is not accessed through any pointer not based on it.

What programs do...

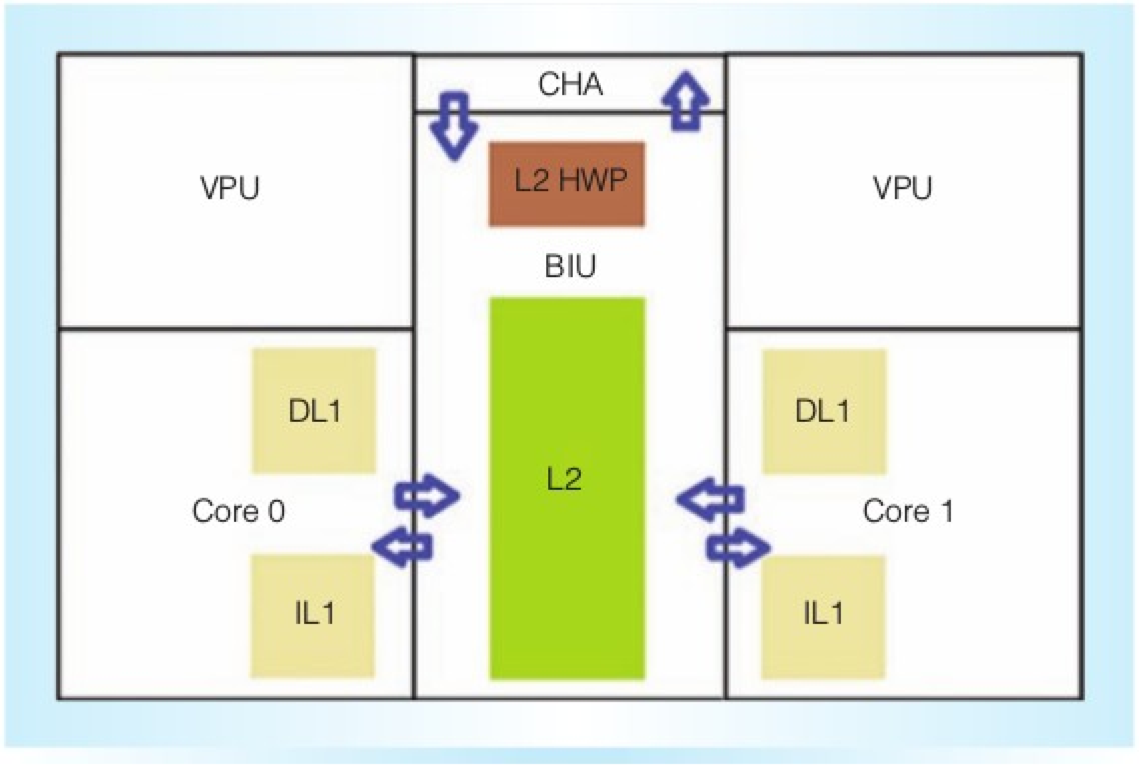

VIM - /big/pkgs/vim-5.2a-cscope/src/main.c

```
/* vi:set ts=8 sts=4 sw=4:
 *
 * VIM - Vi IMproved    by Bram Moolenaar
 *
 * Do ":help uganda"  in Vim to read copying and usage conditions.
 * Do ":help credits" in Vim to see a list of people who contributed.
 */

#define EXTERN
#include "vim.h"

#ifdef SPAWNO
# include <spawno.h>            /* special MSDOS swapping library */
#endif

static void mainerr __ARGS((int, char_u *));
static void usage __ARGS((void));
static int get_number_arg __ARGS((char_u *p, int *idx, int def));

/*
 * Type of error message.  These must match with errors[] in mainerr().
 */
#define ME_UNKNOWN_OPTION       0
                                                        17,4
```



- Read data from memory
- Compute using that data
- Write results back to memory
- Communicate with other nodes and the outside world

Caches

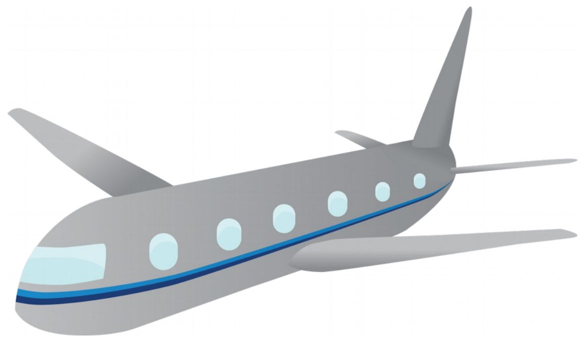


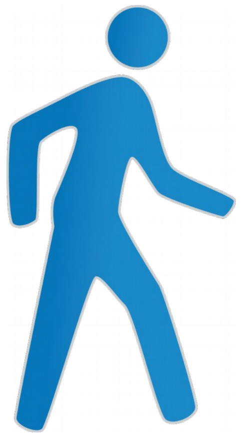
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7453080

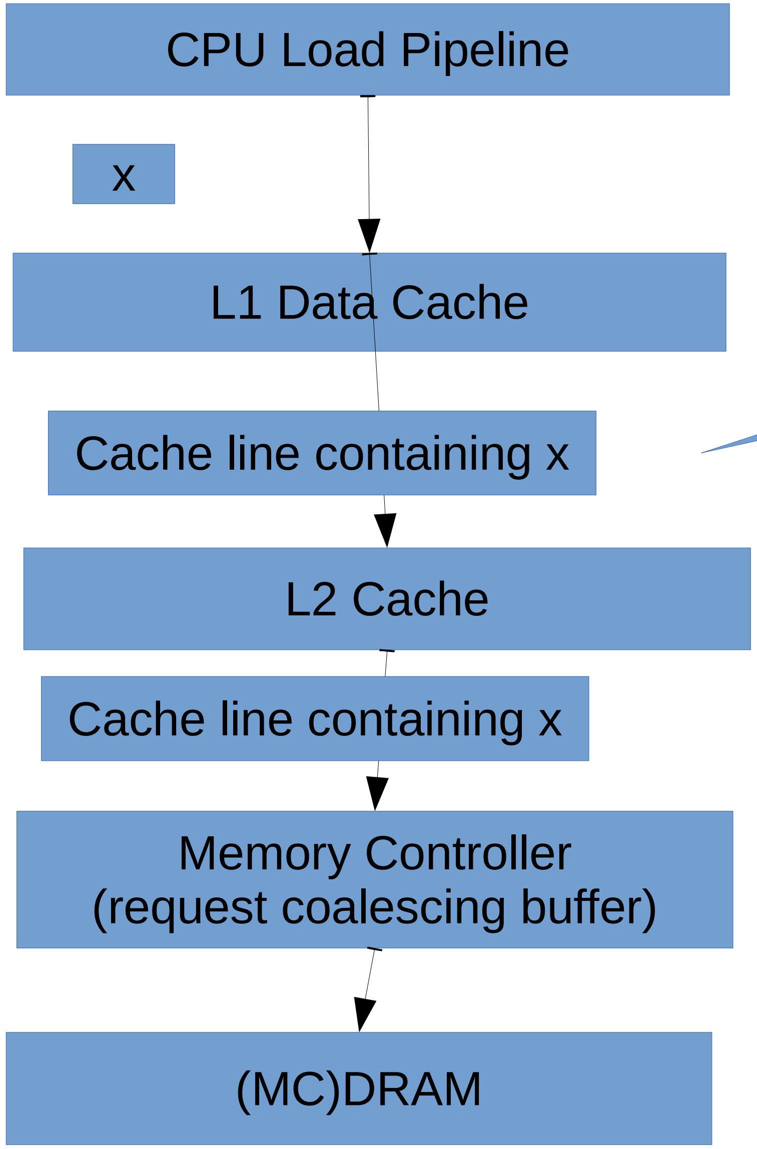
- KNL cores are paired into a "tile", which share an 1 MB L2 cache
- L2 cache can deliver 1 read cache line and 0.5 write cache lines per cycle
- Each core has its own 32 KB L1 I-cache and 32 KB L1 data cache
- The cache is "writeback" - the processor reads a cache line to write to it
- Each cache line is 64 bytes (the size of one 512-bit vector)

NVIDIA Pascal has 64 KB shared memory per SM (see docs on __shared__)

Memory Requests



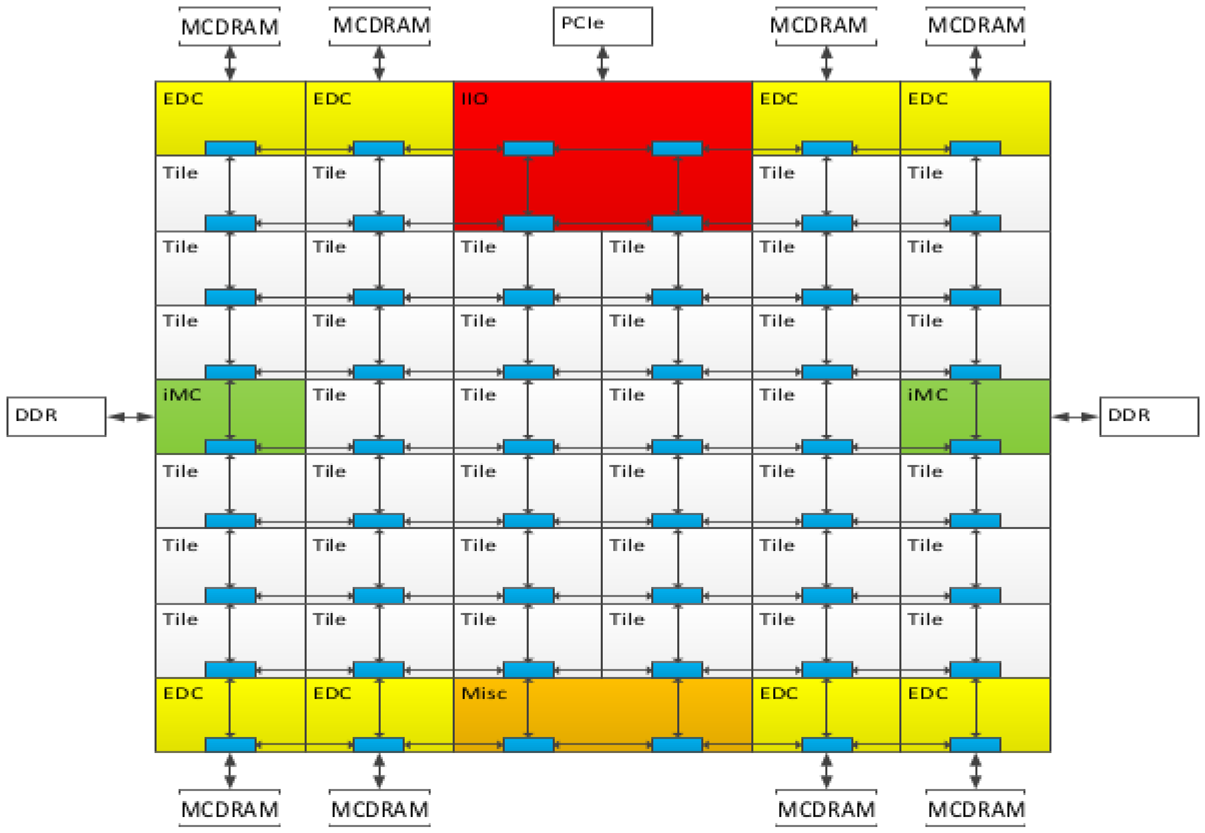




Always fetch whole cache lines (arrange your data accordingly)

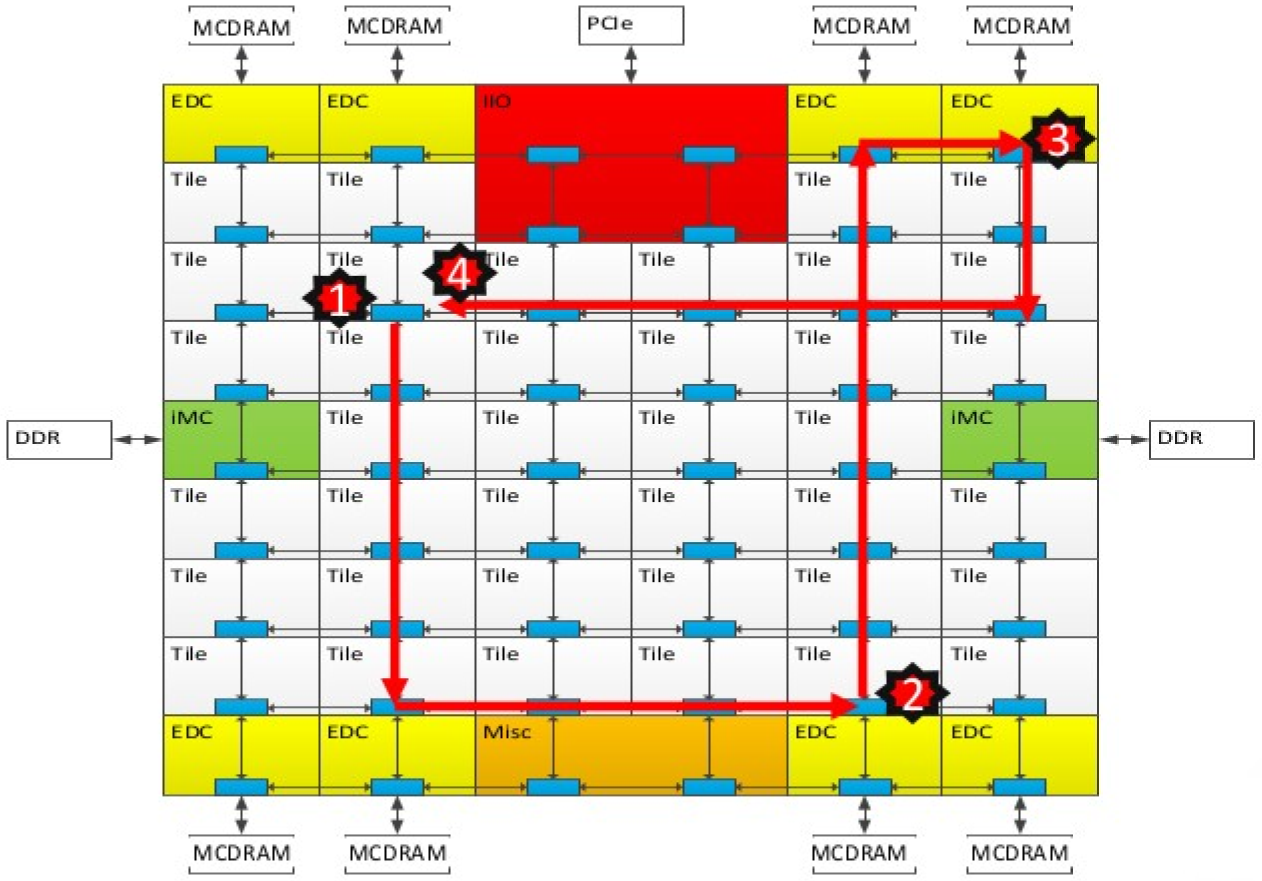
GPUs have coalescing buffers here too, and they can afford to wait longer!

KNL Mesh On-Chip Network



- Tiles are arranged on a mesh
- L2 caches are coherent, so we need tag directories to keep track of which tile owns which cache lines
- How the cache lines are mapped to tag directories has three modes (selected at boot time): all-to-all, quadrant, and sub-NUMA clustering

KNL All-to-all mode



Address uniformly hashed across all distributed directories

No affinity between Tile, Directory and Memory

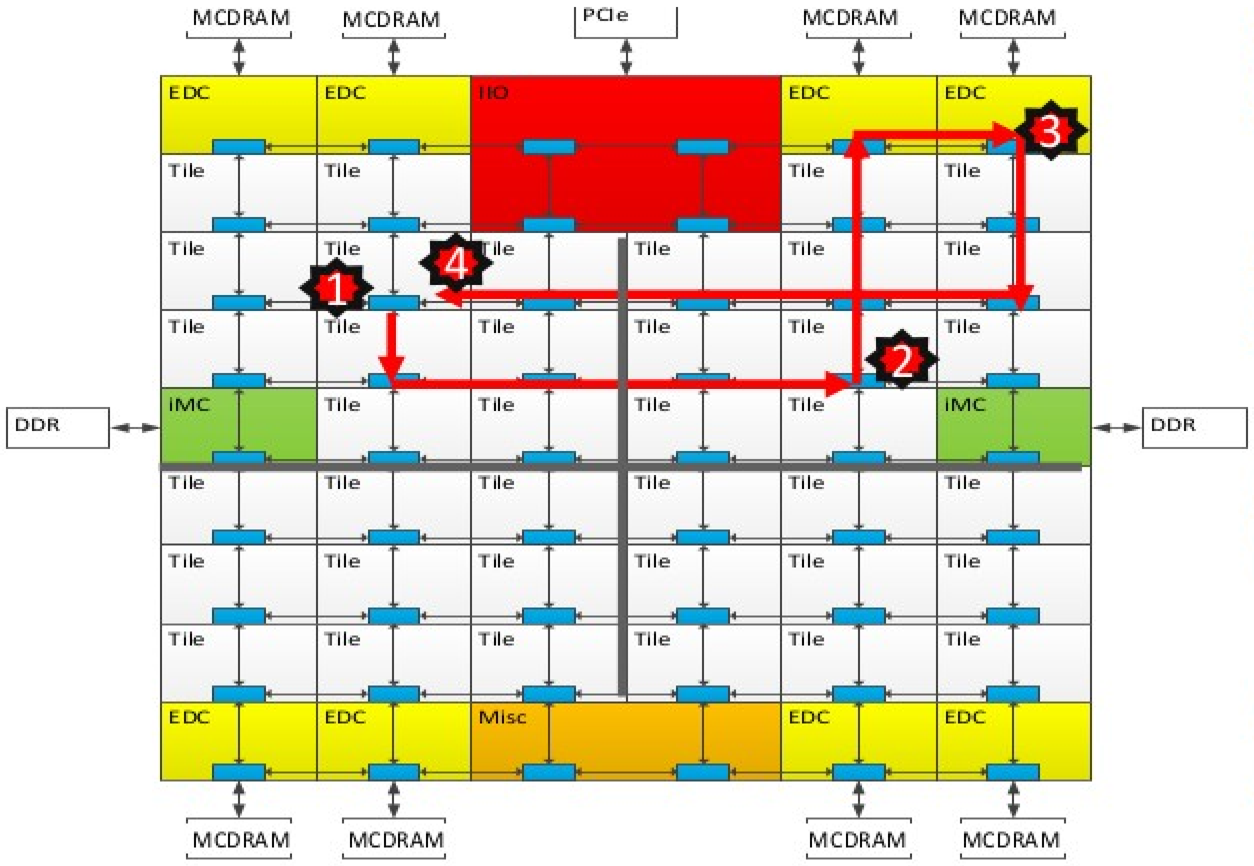
Most general mode. Lower performance than other modes.

Typical Read L2 miss

1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor

Avinash Sodani CGO PPoPP HPCA Keynote 2016

KNL Quadrant Mode



Chip divided into four virtual Quadrants

Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

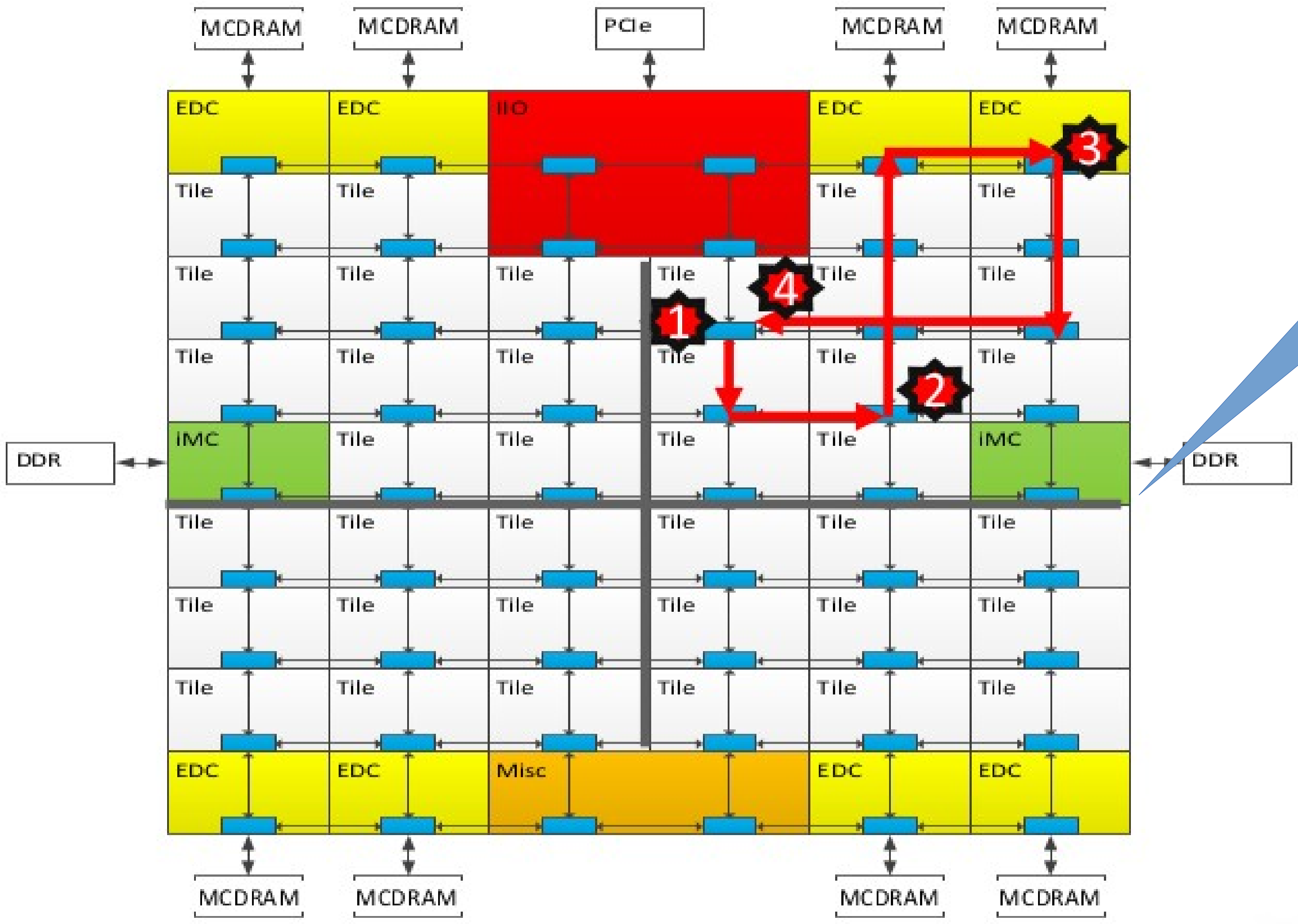
Lower latency and higher BW than all-to-all. SW Transparent.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Avinash Sodani CGO PPoPP HPCA Keynote 2016

KNL Sub-NUMA Clustering (SNC) Mode

Note, however, that quadrants are not symmetric!



1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Each Qua (Cluster) exposed as a separa UMA domain to OS.

L oks analogous to 4-Socket Xeon

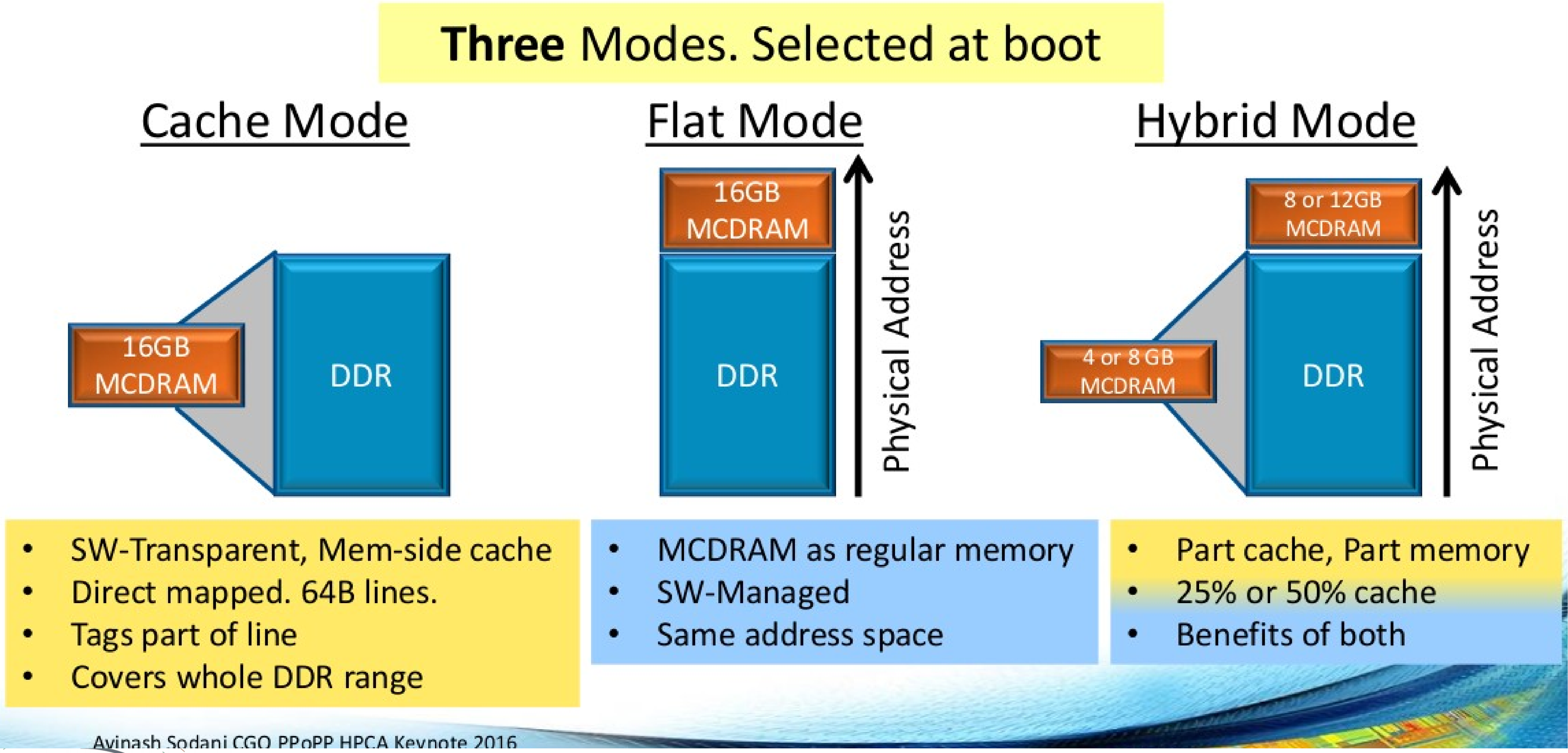
Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

SW needs to NUMA optimize to get benefit.

Run one MPI rank per quadrant?

HBM Modes



Three Modes. Selected at boot

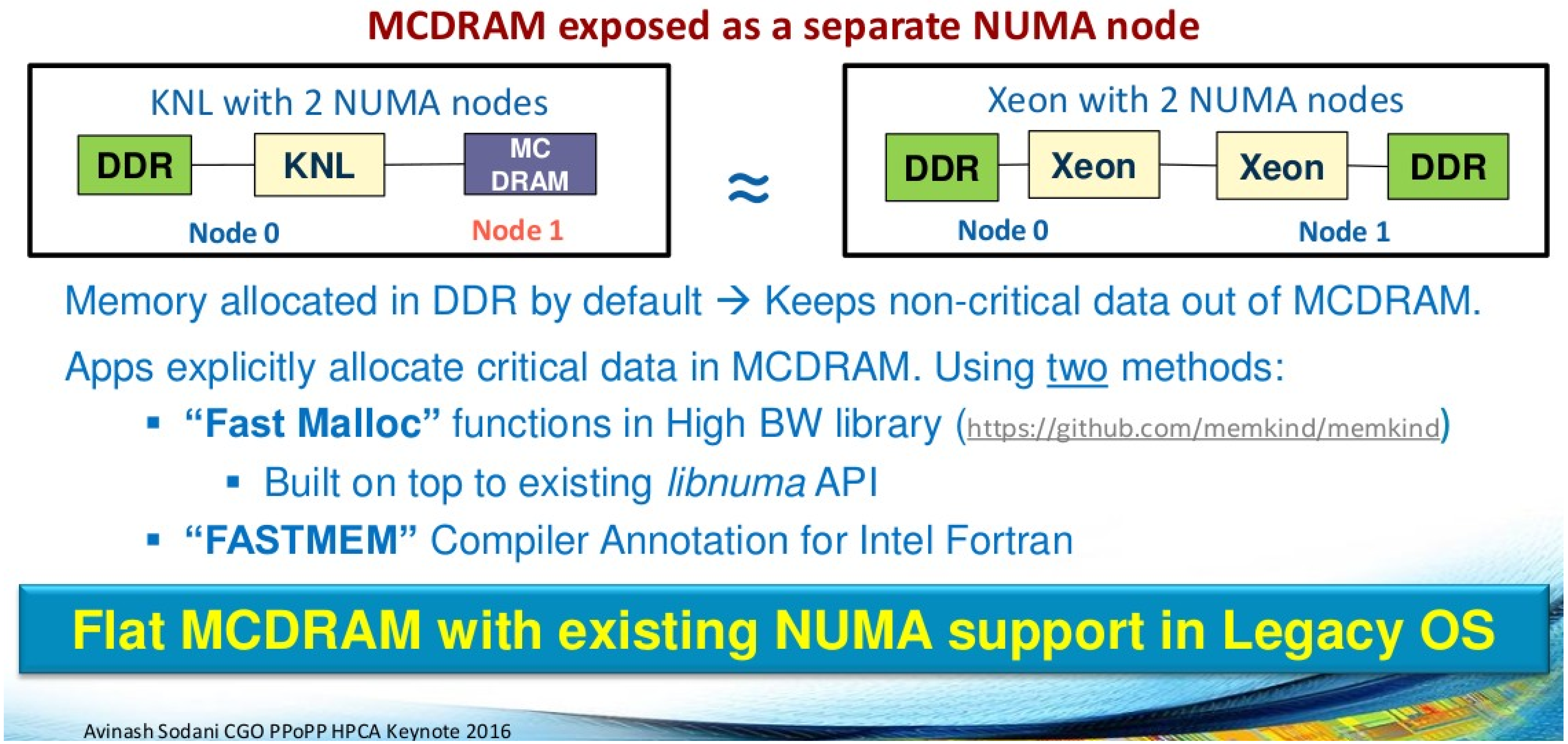
- SW-Transparent, Mem-side cache
- Direct mapped. 64B lines.
- Tags part of line
- Covers whole DDR range

- MCDRAM as regular memory
- SW-Managed
- Same address space

- Part cache, Part memory
- 25% or 50% cache
- Benefits of both

Avinash Sodani CGO PPoPP HPCA Keynote 2016

How Flat Mode Looks



MCDRAM exposed as a separate NUMA node

KNL with 2 NUMA nodes
DDR
KNL
MC DRAM
Node 0
Node 1
≈
Xeon with 2 NUMA nodes
DDR
Xeon
Xeon
DDR
Node 0
Node 1

Memory allocated in DDR by default → Keeps non-critical data out of MCDRAM.

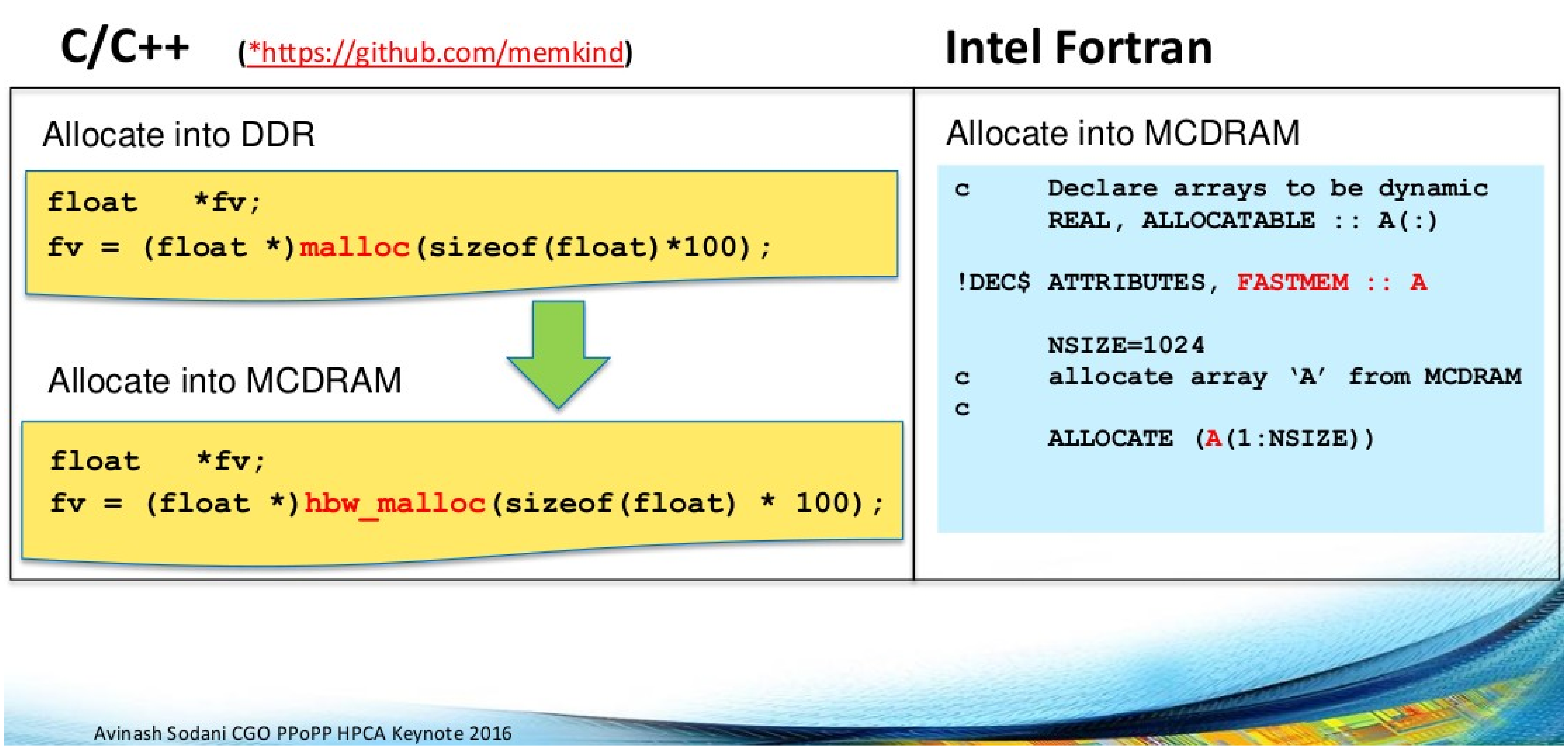
Apps explicitly allocate critical data in MCDRAM. Using two methods:

- **"Fast Malloc"** functions in High BW library (https://github.com/memkind/memkind)
  - Built on top to existing *libnuma* API
- **"FASTMEM"** Compiler Annotation for Intel Fortran

Flat MCDRAM with existing NUMA support in Legacy OS

Avinash Sodani CGO PPoPP HPCA Keynote 2016

Flat Mode Memory Management



C/C++ (*https://github.com/memkind)

Allocate into DDR

```
float    *fv;
fv = (float *)malloc(sizeof(float)*100);
```

Allocate into MCDRAM

```
float    *fv;
fv = (float *)hbw_malloc(sizeof(float) * 100);
```

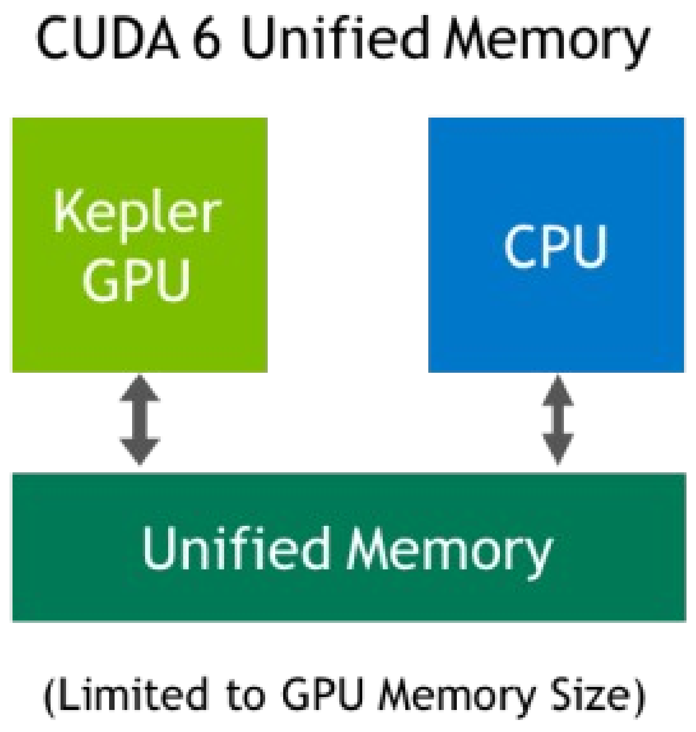
Intel Fortran

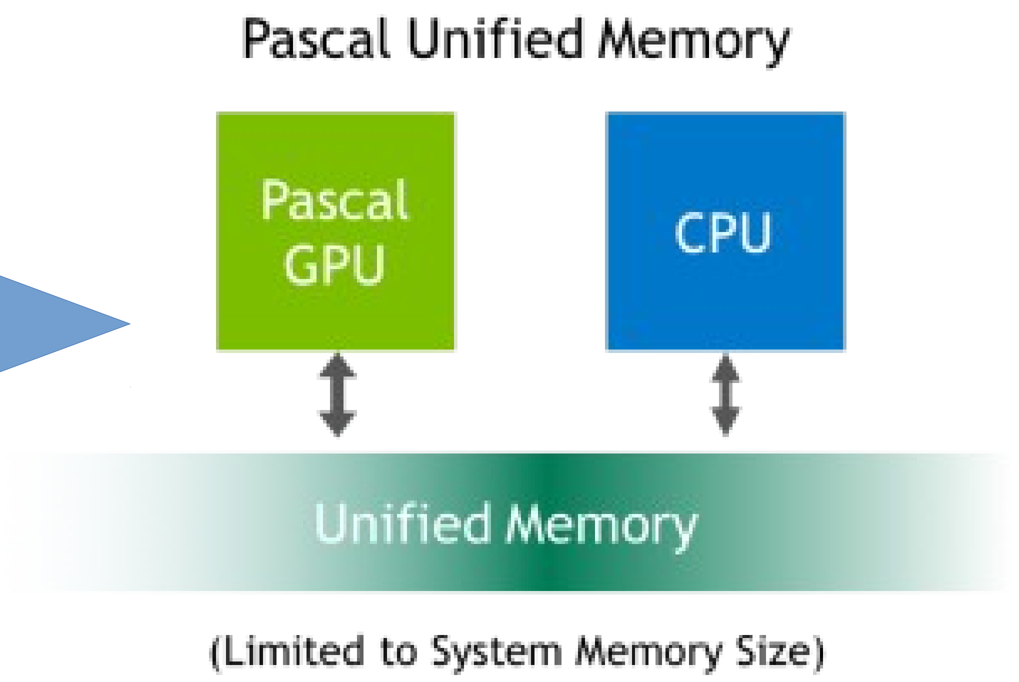
Allocate into MCDRAM

```
c       Declare arrays to be dynamic
        REAL, ALLOCATABLE :: A(:)

!DEC$ ATTRIBUTES, FASTMEM :: A

        NSIZE=1024
c       allocate array 'A' from MCDRAM
c
        ALLOCATE (A(1:NSIZE))
```

Avinash Sodani CGO PPoPP HPCA Keynote 2016

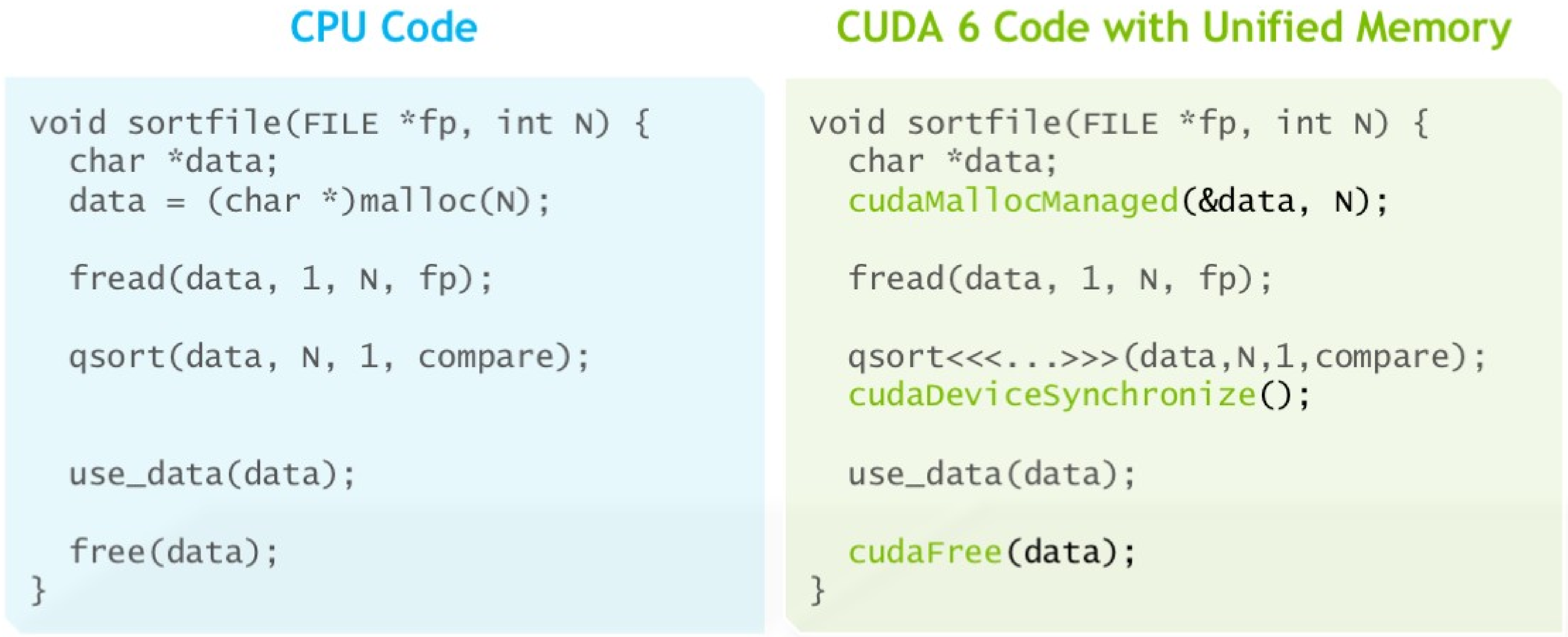
CUDA Unified Memory





Unified memory enables “lazy” transfer on demand – will mitigate/eliminate the “deep copy” problem!

CUDA UM (The Old Way)



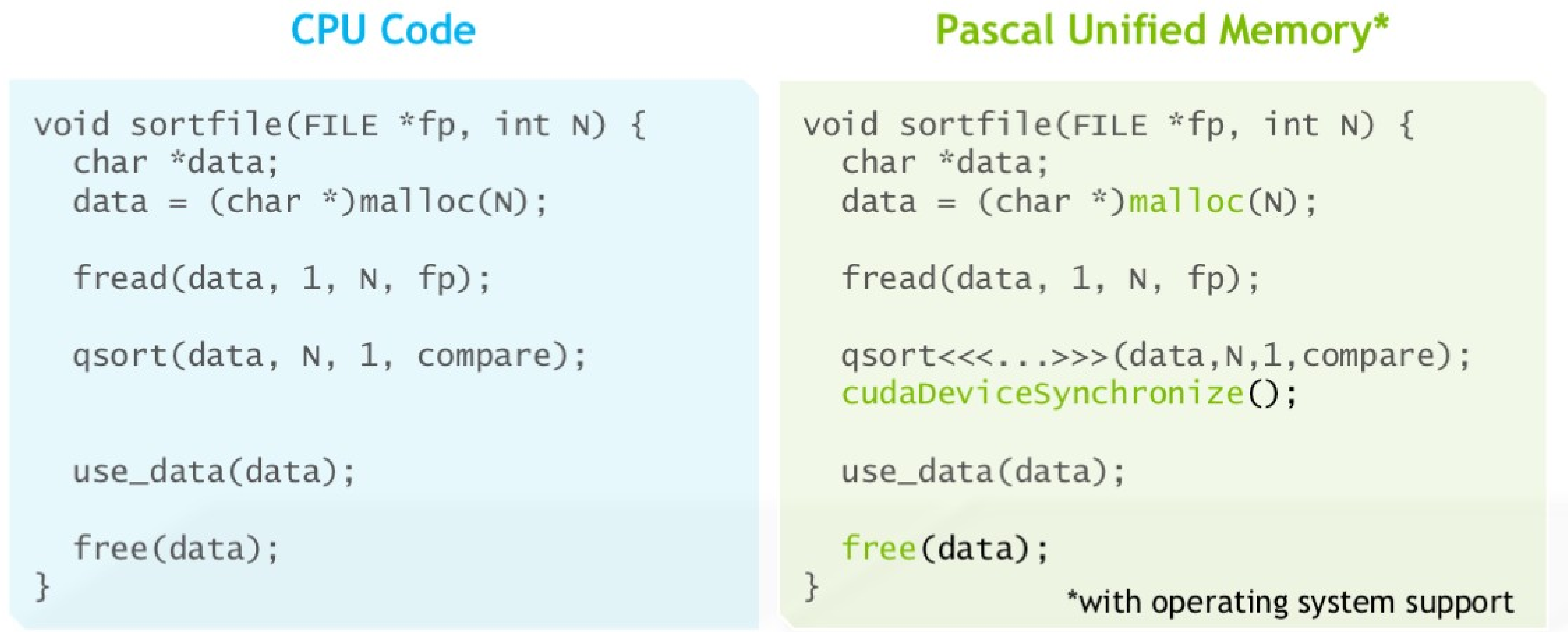
CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

CUDA UM (The New Way)



CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

Pascal Unified Memory*

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  free(data);
}
```

*with operating system support

Pointers are “the same” everywhere!

Types of parallelism

- ✓ Parallelism across nodes (using MPI, etc.)
- ✓ Parallelism across sockets within a node [Not applicable to the BG/Q, KNL, etc.]
- ✓ Parallelism across cores within each socket
- ✓ Parallelism across pipelines within each core (i.e. instruction-level parallelism)
- ✓ Parallelism across vector lanes within each pipeline (i.e. SIMD)
- ✓ Using instructions that perform multiple operations simultaneously (e.g. FMA)

Hardware threads tie in here too!

How fast can you go...

The speed at which you can compute is bounded by:

(the clock rate of the cores) x (the amount of parallelism you can exploit)

BG/Q: Fixed 1.66 GHz
KNL: 1.30 GHz
(dynamically scaled)

Kepler: 0.8 GHZ
Pascal: 1.30 GHz

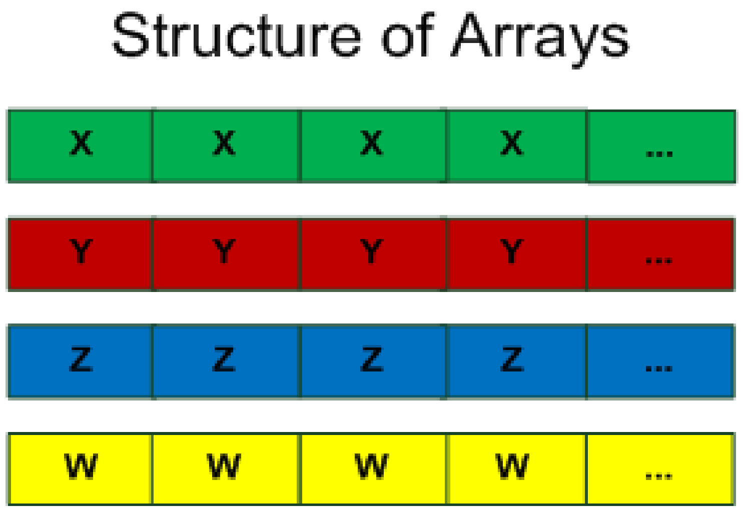
Your hard work goes here...

Hardware Prefetcher

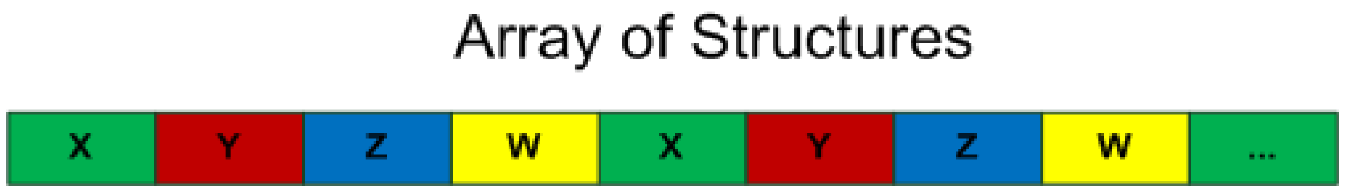
- L1 hardware prefetcher monitors access patterns and generates requests to the L2 in advance of anticipated need
- L2 hardware prefetcher does the same, issuing requests to main memory
- The KNL L2 prefetcher supports 48 independent streams (that's shared among all running threads). Running 4 hardware threads per core, two cores per tile: 6 streams per thread!

AOS vs. SOA

Easy to vectorize; uses lots of prefetching streams!



```
struct Particles {
  float *x;
  float *y;
  float *z;
  float *w;
};
```



```
struct Particle {
  float x;
  float y;
  float z;
  float w;
};

struct Particle *Particles;
```

https://software.intel.com/en-us/articles/ticker-tape-part-2

Better cache locality; fewer prefetcher streams
with scatter/gather support, maybe vectorization is not so bad!

Compiling

Basic optimization flags...

- ✓ -O3 – Generally aggressive optimizations (try this first)
- ✓ -g – Always include debugging symbols (**really, always**! - when your run crashes at scale after running for hours, you want the core file to be useful)
- ✓ -fopenmp – Enable OpenMP (the pragmas will be ignored without this)
- ✓ -ffast-math (clang, gcc, etc.) – Enable “fast” math optimizations (most people don't need strict IEEE floating-point semantics).

If you don't use -O<n> to turn on some optimizations, most of the previous material is irrelevant!

An example... (what tuning might look like)

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel for
        for (i = 0; i < n; ++i) {
                a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
                m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
        }
}
```

We use restrict here to tell the compiler that the arrays are disjoint in memory.

We likely want at least 2 threads per core, probably 4.

Each statement requires 5 prefetcher streams, on some systems this is too many...

Split the loop

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel for
        for (i = 0; i < n; ++i) {
                a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
        }
#pragma omp parallel for
        for (i = 0; i < n; ++i) {
                m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
        }
}
```

We could also change the data structures being used so that we have arrays of structures (although that might inhibit vectorization).

An example...

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel for
        for (i = 0; i < n; ++i) {
                a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
        }
#pragma omp parallel for
        for (i = 0; i < n; ++i) {
                m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
        }
}
```

We did a bit too much splitting here (starting each of these parallel regions can be expensive).

(don't actually split the parallel region)

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel
  {
#pragma omp for
        for (i = 0; i < n; ++i) {
                a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
        }
#pragma omp for
        for (i = 0; i < n; ++i) {
                m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
        }
  }
}
```

An example...

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel
  {
#pragma omp for
       for (i = 0; i < n; ++i) {
              a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
       }
       ...
}
```

On PowerPC, the RHS expression is two dependent FMAs requiring at least 3 vector registers (5 registers if we “preload” all of the input values). On X86, with implicit memory operands, we require fewer. The first FMA has a 6-cycle latency, and if we run two threads/core, we have an effective latency of 3 cycles/thread to hide.

Unroll (interleaved) by a factor of 3. On PowerPC, this will require up to 3*5 == 15 vector registers, but we have 32 of them.

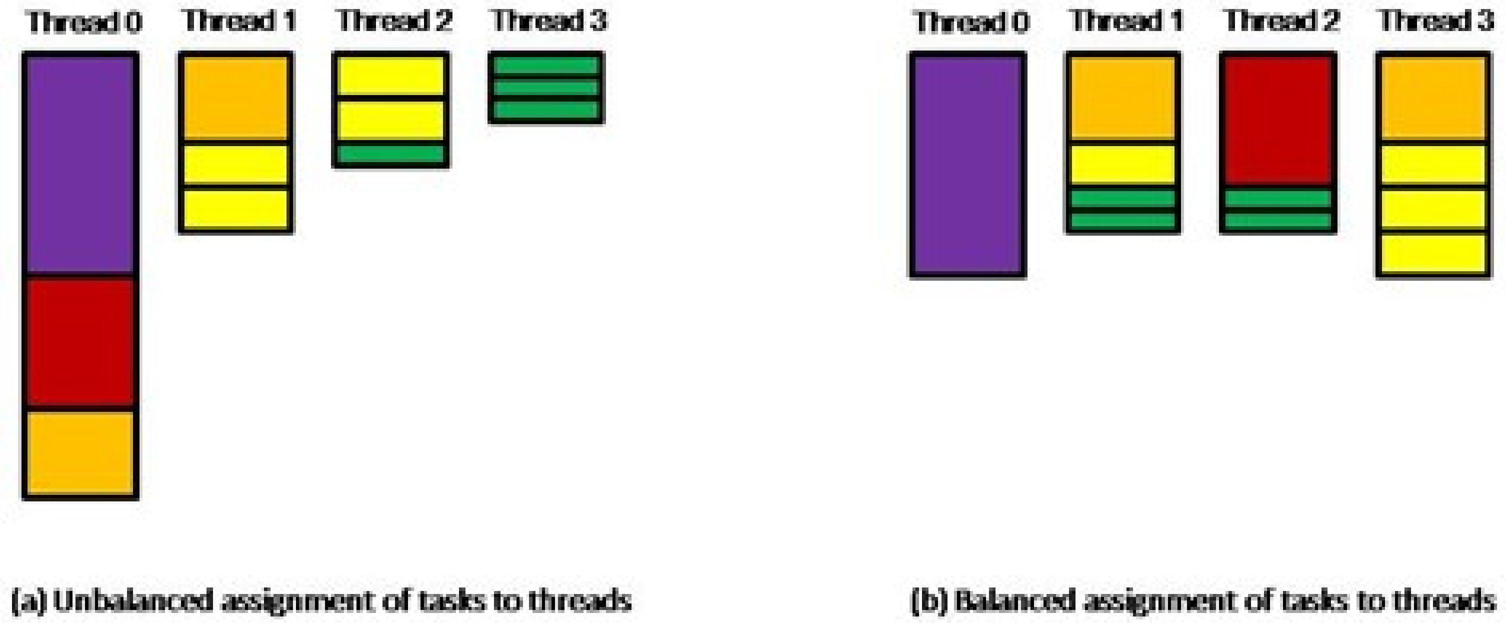
```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel
  {
#pragma omp for
#pragma unroll(3)
       for (i = 0; i < n; ++i) {
              a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
       }
       ...
```

The compiler should do this automatically, if profitable, but in case it doesn't...

schedule(dynamic) can be your friend...

```
#pragma omp parallel for schedule(dynamic)
  for (i = 0; i < n; i++) {
    unknown_amount_of_work(i);
  }
```

You can use schedule(dynamic, <n>) to distribute in chunks of size n.



(a) Unbalanced assignment of tasks to threads

(b) Balanced assignment of tasks to threads

https://software.intel.com/en-us/articles/load-balance-and-parallel-performance

#pragma omp simd

Starting with OpenMP 4.0, OpenMP also supports explicit vectorization...

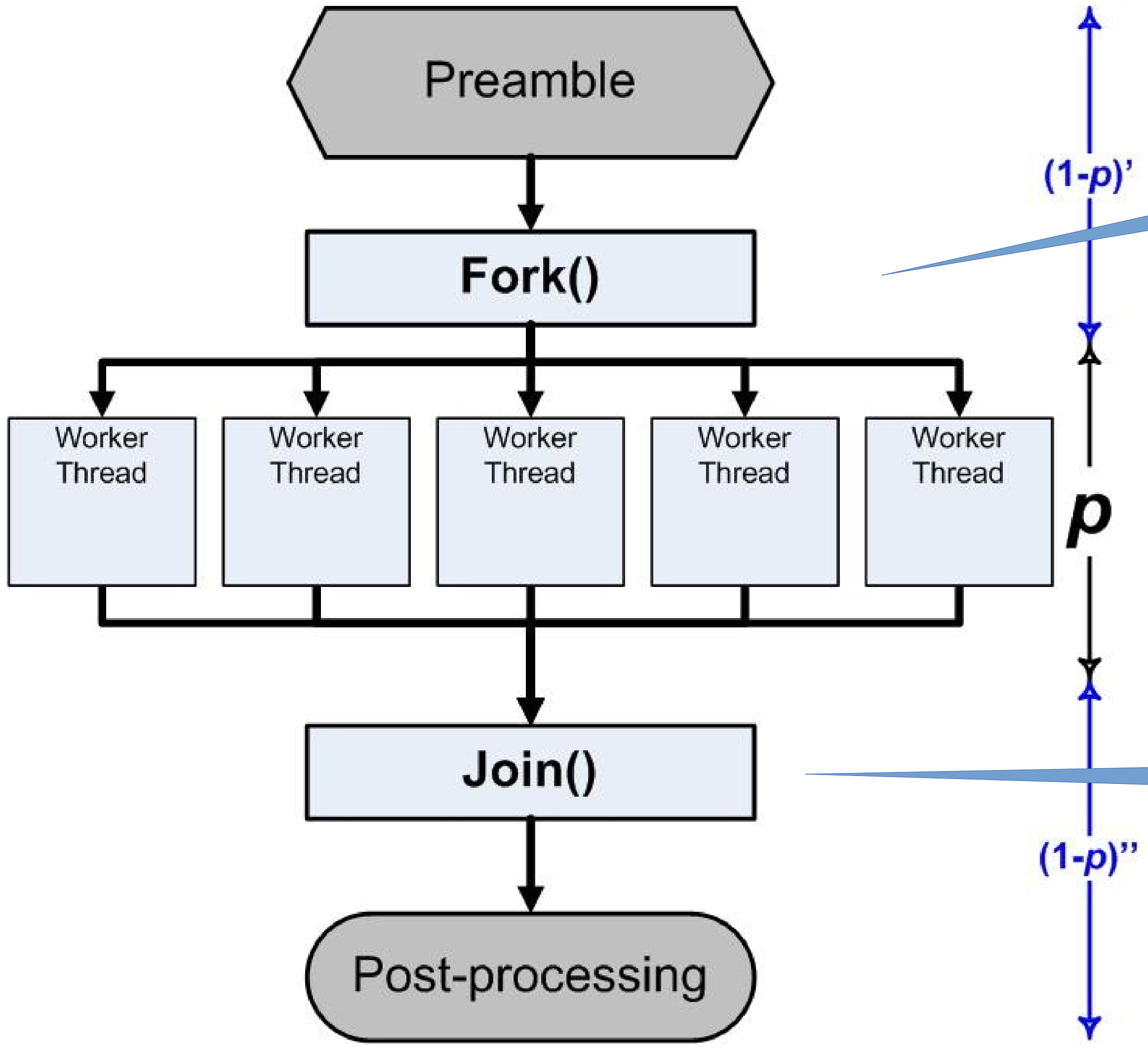
```
char foo(char *A, int n) {
  int i;
  char x = 0;
#pragma omp simd reduction(+:x)
  for (i=0; i<n; i++){
    x = x + A[i];
  }
  return x;
}
```

Can combine with threading...

```
char foo(char *A, int n) {
  int i;
  char x = 0;
#pragma omp parallel for simd reduction(+:x)
  for (i=0; i<n; i++){
    x = x + A[i];
  }
  return x;
}
```

https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

Coarse Grained vs. Fine Grained Parallelism



This is expensive (many thousands of cycles)

This is expensive too!

Amdahl's law says the speedup is limied to: 1/(1 – p). So if 5% of the program remains serial, then the speedup from parallelization is limited to 20x.

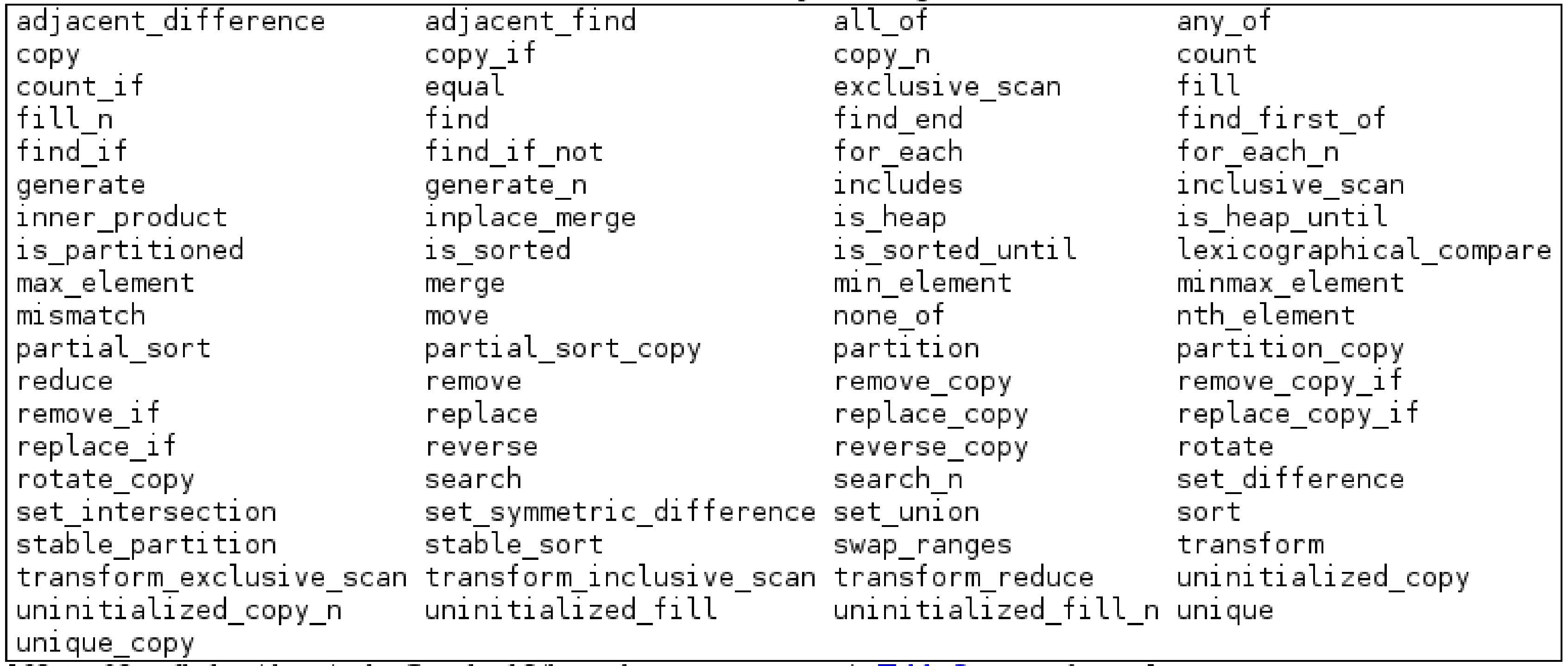
https://en.wikipedia.org/wiki/Amdahl%27s_law

https://blogs.msdn.microsoft.com/ddperf/2009/04/29/parallel-scalability-isnt-childs-play-part-2-amdahls-law-vs-gunthers-law/

C++17 Parallel Algorithms

- Parallel versions, and parallel+vectorized versions, of almost all standard algorithms (plus a few new ones)

Table 2 — Table of parallel algorithms



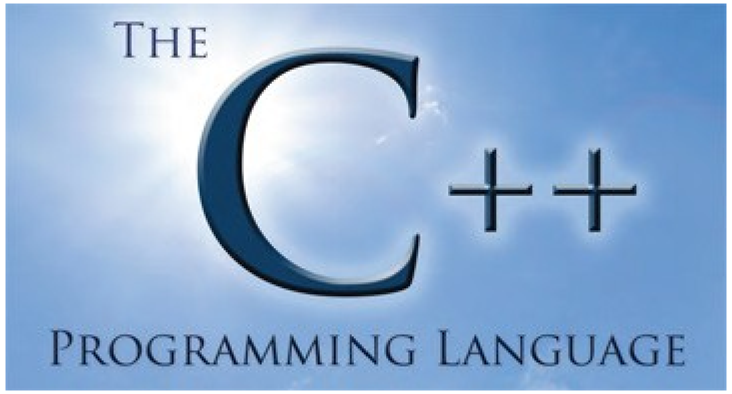
adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

[*Note:* Not all algorithms in the Standard Library have counterparts in Table 2. — *end note*]

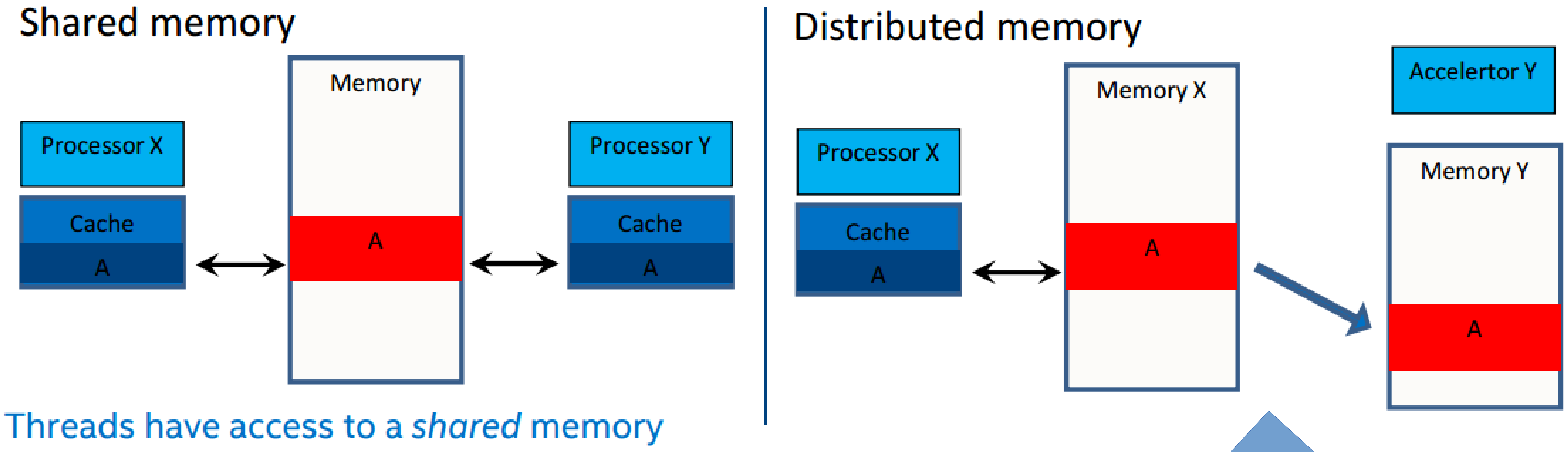
C++17 Parallel Algorithms

```
vector<float> a;
…
for_each(par_seq, a.begin(), a.end(), [&](float &f) {
  f += 2.0;
});
```

Coming soon to a compiler near you!



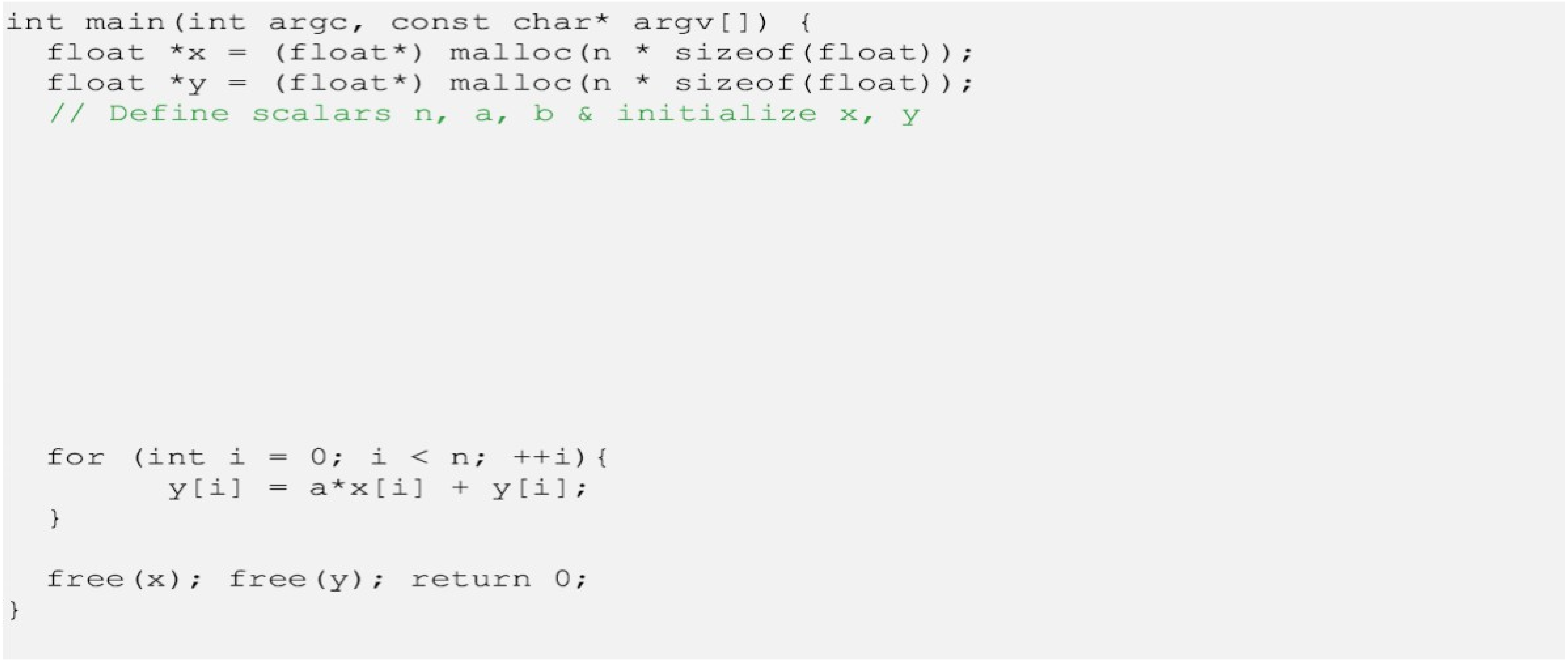
OpenMP Evolving Toward Accelerators



http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

New in OpenMP 4

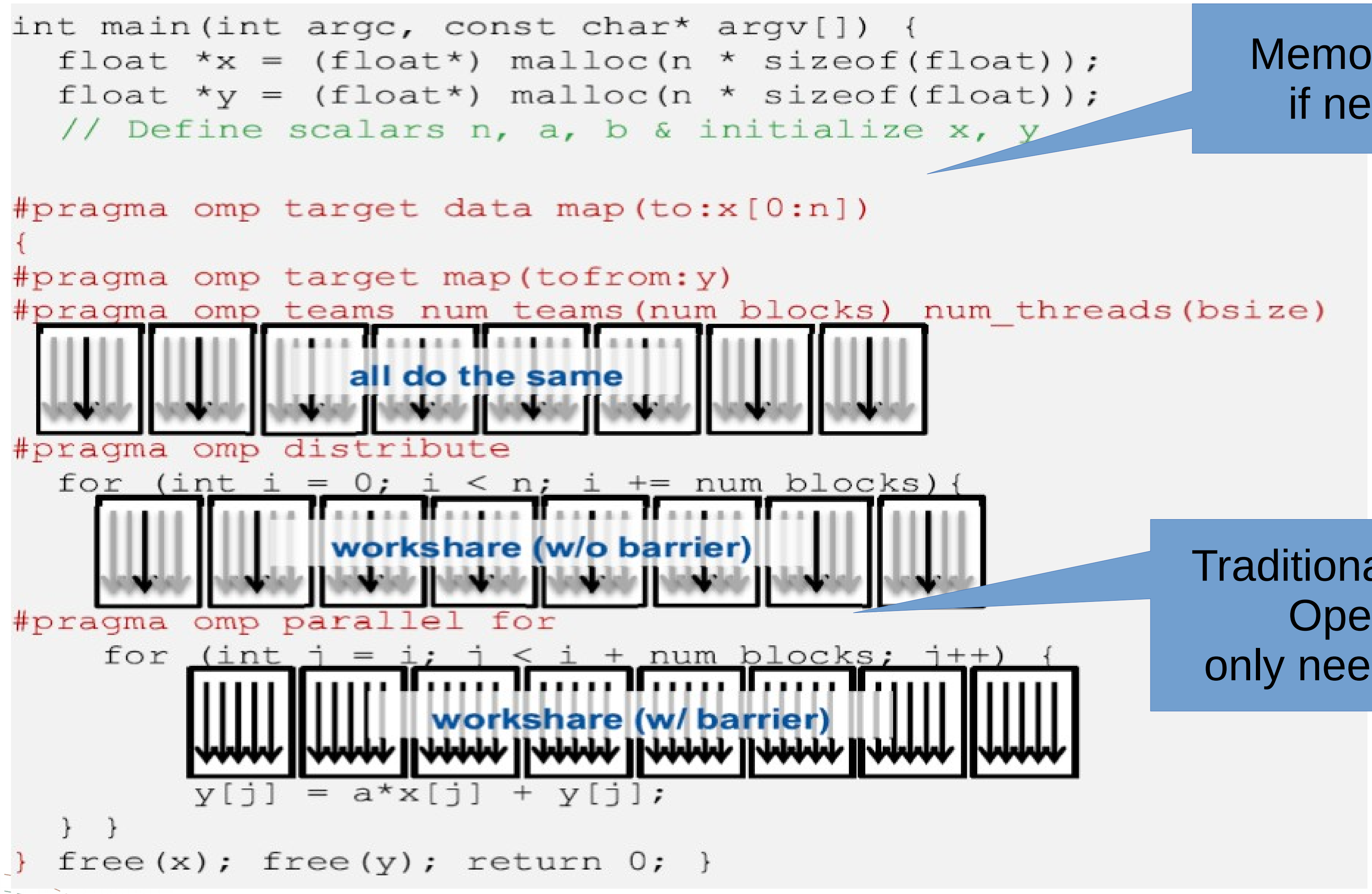
OpenMP Accelerator Support – An Example (SAXPY)



```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y









  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

http://llvm-hpc2-workshop.github.io/slides/Wong.pdf

OpenMP Accelerator Support – An Example (SAXPY)



```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
      y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

Memory transfer if necessary.

Traditional CPU-targeted OpenMP might only need this directive!

OpenMP and UVM?

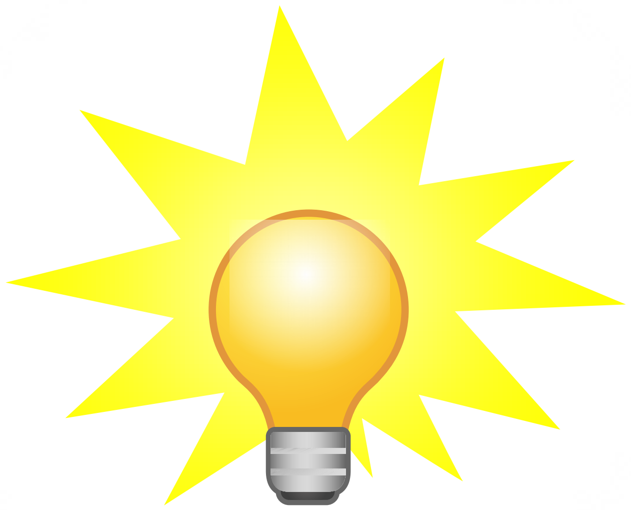
How does OpenMP accelerator support interact with unified memory?

We don't yet know!

MKL, cuBLAS, ESSL, etc.

Vendors provide optimized math libraries for each system (BLAS for linear algebra, FFTs, and more).

- MKL on Intel systems, ESSL on IBM systems, cuBLAS (and others) for NVIDIA GPUs
- For FFTs, there is often an optional FFTW-compatible interface.



Memory partitioning

Using threads vs. multiple MPI ranks per node: it's about...

- Memory
  - Sending data between ranks on the same node often involves “unnecessary” copying (unless using MPI-3 shared memory windows)
  - Similarly, your application may need to manage “unnecessary” ghost regions
  - MPI (and underlying components) have data structures that grow linearly (at best) with the total number of ranks
- And Memory
  - When threads can work together they can share resources instead of competing (cache, memory bandwidth, etc.)
  - Each process only gets a modest amount of memory per core
- And parallelism
  - You'll likely see the best overall results from the scheme that exposes the most parallelism

And finally, be kind to the file system...

- ✓ Use MPI I/O - use collective I/O if the amounts being written are small
- ✓ Give each rank its own place within the file to store its data (avoid lock contention)
- ✓ Make sure you can validate your data (use CRCs, etc.), and then actually validate it when you read it

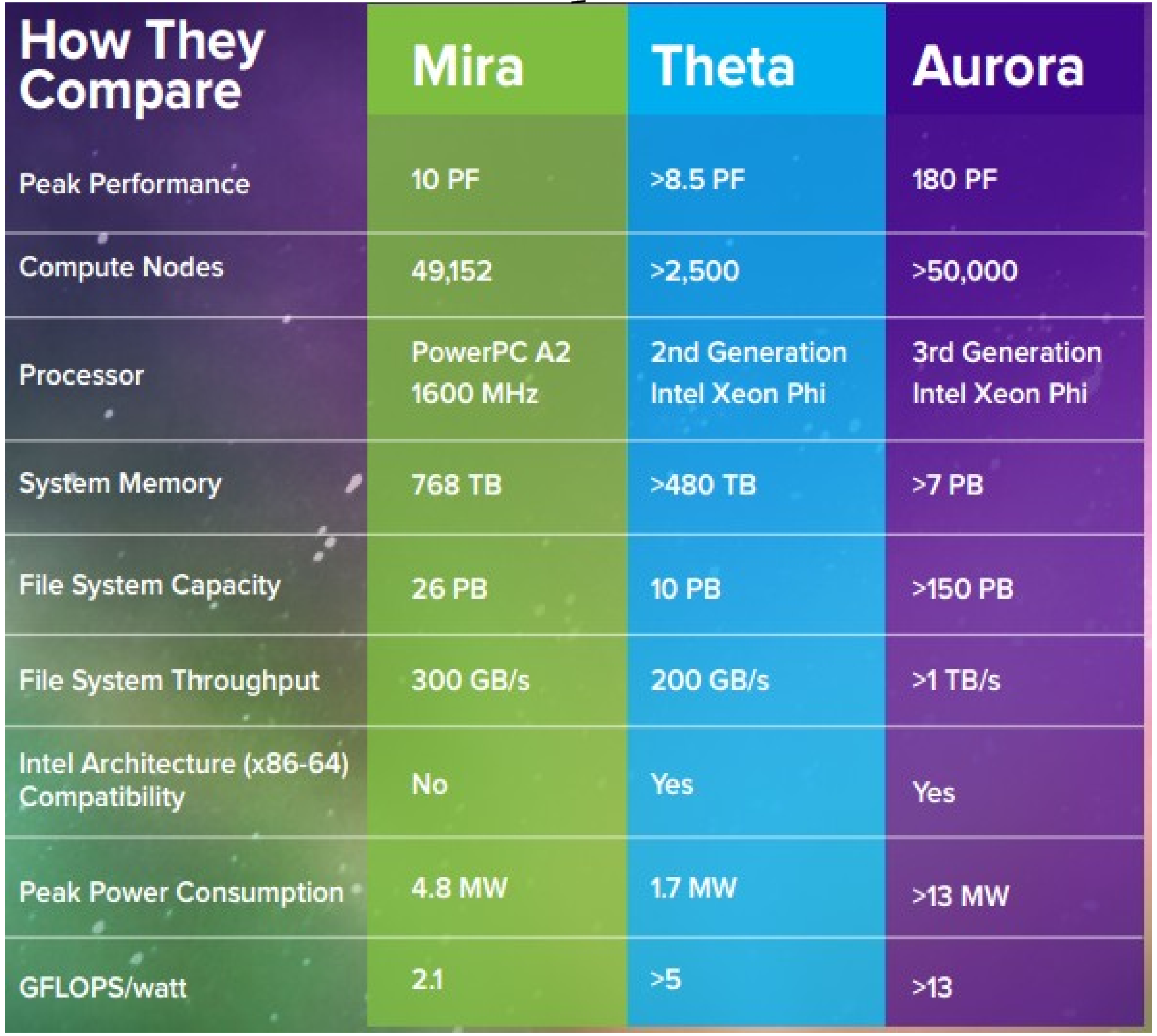
  (We've open-sourced a library for computing CRCs: http://trac.alcf.anl.gov/projects/hpcrc64/)

You probably want to design your files to be write optimized, not read optimized! Why?
You generally write more checkpoints than you read (and time reading from smaller jobs is "free").
And writing is slower than reading.

And use load + broadcast instead of reading the same thing from every rank...

- ✓ Static linking is the default for all IBM BG/Q compilers for good reason... loading shared libraries from tens of thousands of ranks may not be fast
- ✓ The same is true for programs using embedded scripting languages... loading lots of small script files from tens of thousands of ranks is even worse

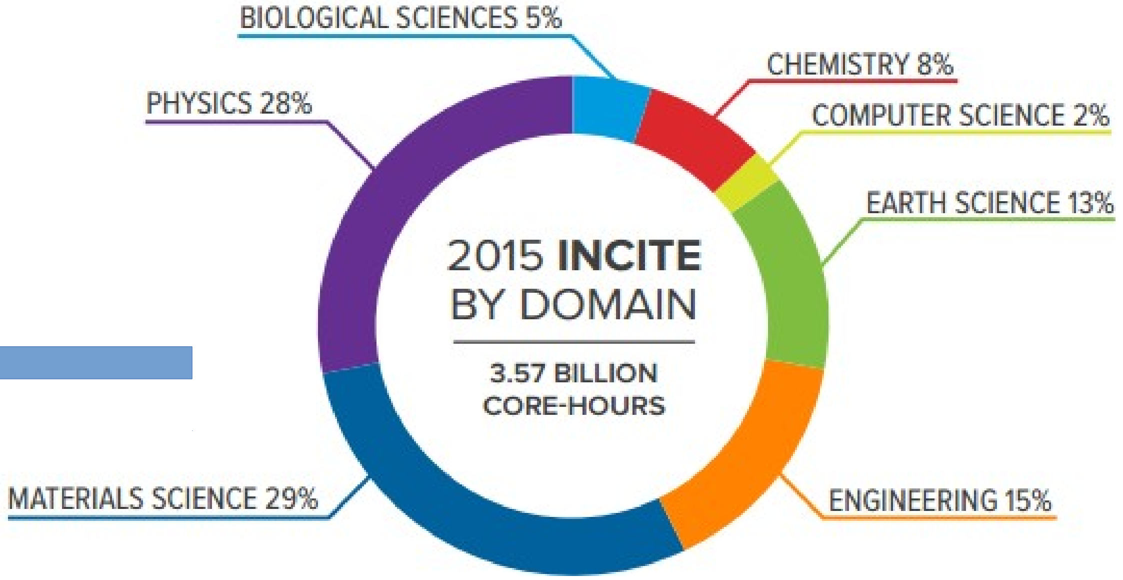
ALCF Systems

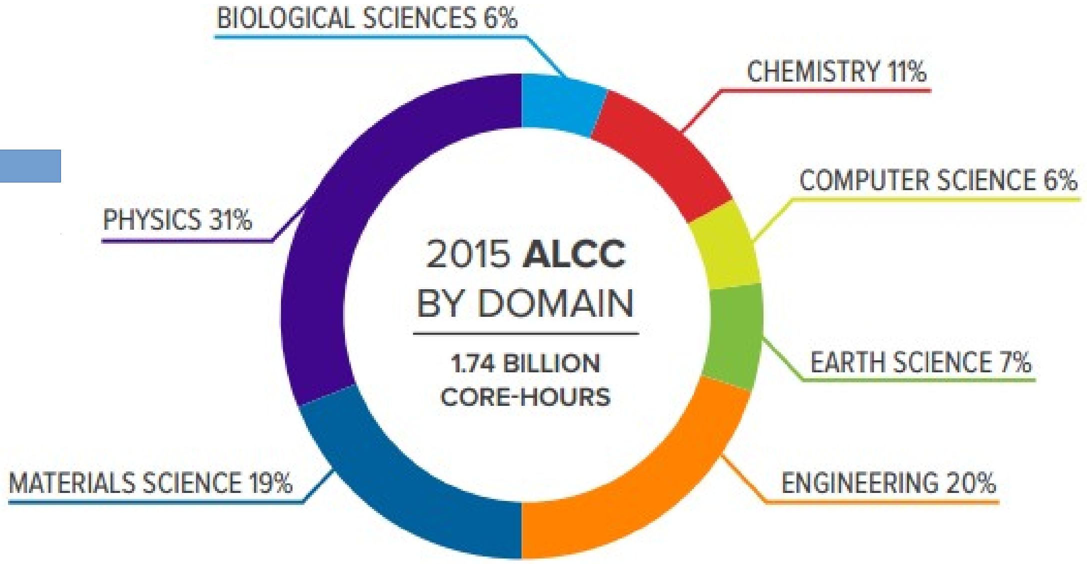


How They Compare	Mira	Theta	Aurora
Peak Performance	10 PF	>8.5 PF	180 PF
Compute Nodes	49,152	>2,500	>50,000
Processor	PowerPC A2 1600 MHz	2nd Generation Intel Xeon Phi	3rd Generation Intel Xeon Phi
System Memory	768 TB	>480 TB	>7 PB
File System Capacity	26 PB	10 PB	>150 PB
File System Throughput	300 GB/s	200 GB/s	>1 TB/s
Intel Architecture (x86-64) Compatibility	No	Yes	Yes
Peak Power Consumption	4.8 MW	1.7 MW	>13 MW
GFLOPS/watt	2.1	>5	>13

https://www.alcf.anl.gov/files/alcfscibro2015.pdf

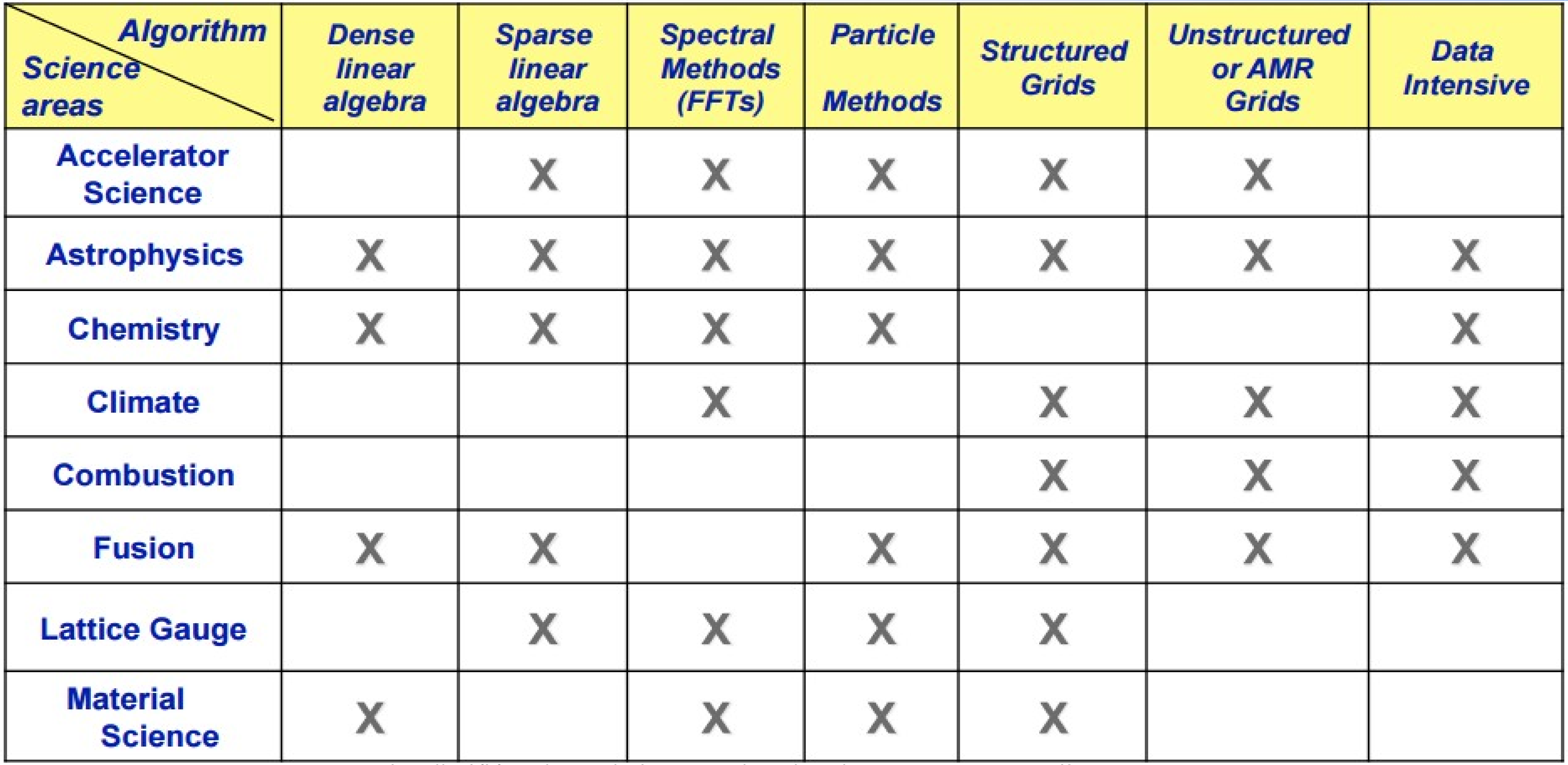
Mira by Domain





https://www.alcf.anl.gov/files/alcfscibro2015.pdf

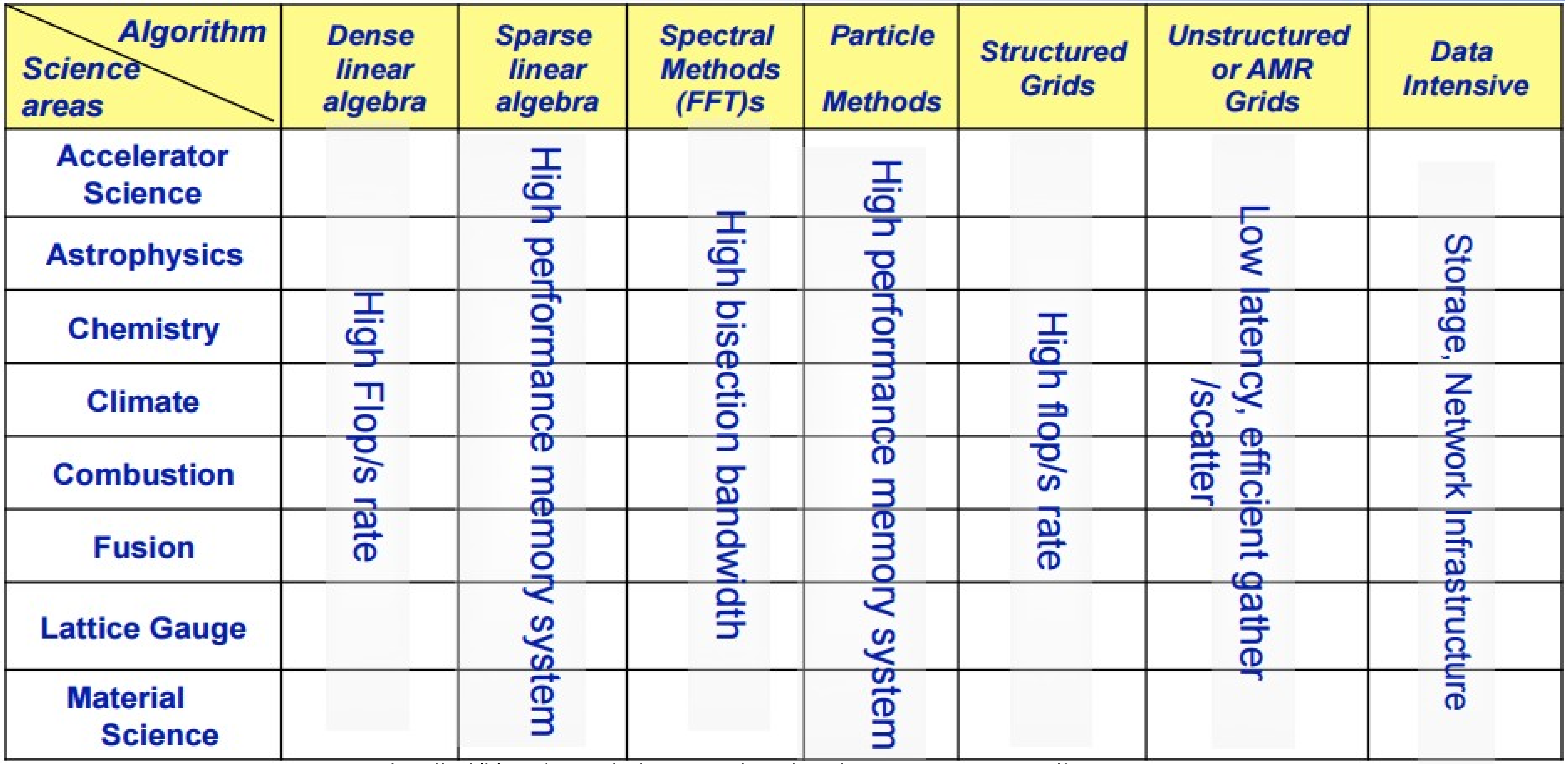
Common Algorithm Classes in HPC



Algorithm / Science areas	Dense linear algebra	Sparse linear algebra	Spectral Methods (FFTs)	Particle Methods	Structured Grids	Unstructured or AMR Grids	Data Intensive
Accelerator Science		X	X	X	X	X	
Astrophysics	X	X	X	X	X	X	X
Chemistry	X	X	X	X			X
Climate			X		X	X	X
Combustion					X	X	X
Fusion	X	X		X	X	X	X
Lattice Gauge		X	X	X	X		
Material Science	X		X	X	X		

http://crd.lbl.gov/assets/pubs_presos/CDS/ATG/WassermanSOTON.pdf

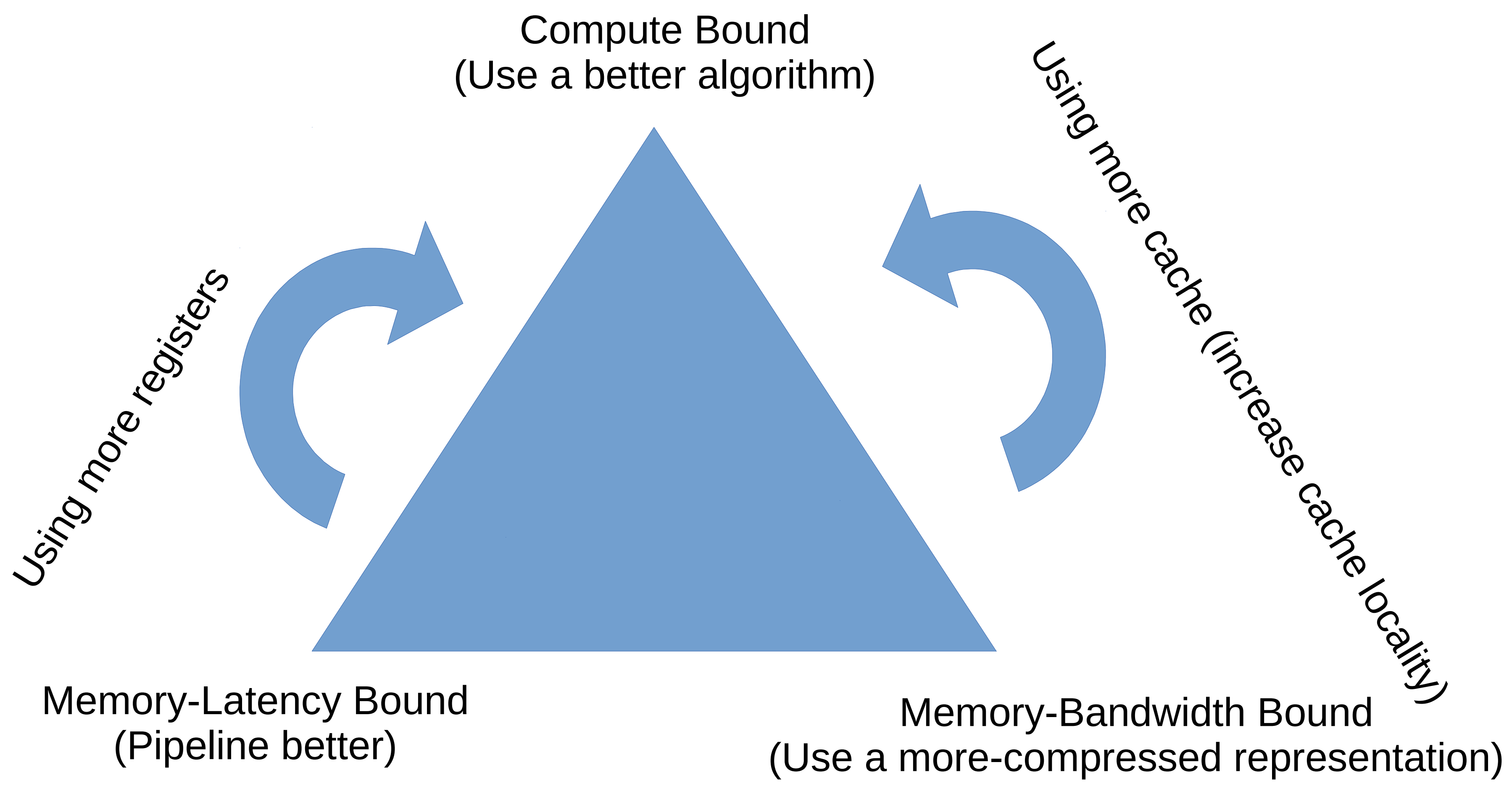
Common Algorithm Classes in HPC – What do they need?



Algorithm Science areas	Dense linear algebra	Sparse linear algebra	Spectral Methods (FFT)s	Particle Methods	Structured Grids	Unstructured or AMR Grids	Data Intensive
Accelerator Science	High Flop/s rate	High performance memory system	High bisection bandwidth	High performance memory system	High flop/s rate	Low latency, efficient gather /scatter	Storage, Network Infrastructure
Astrophysics							
Chemistry							
Climate							
Combustion							
Fusion							
Lattice Gauge							
Material Science							

http://crd.lbl.gov/assets/pubs_presos/CDS/ATG/WassermanSOTON.pdf

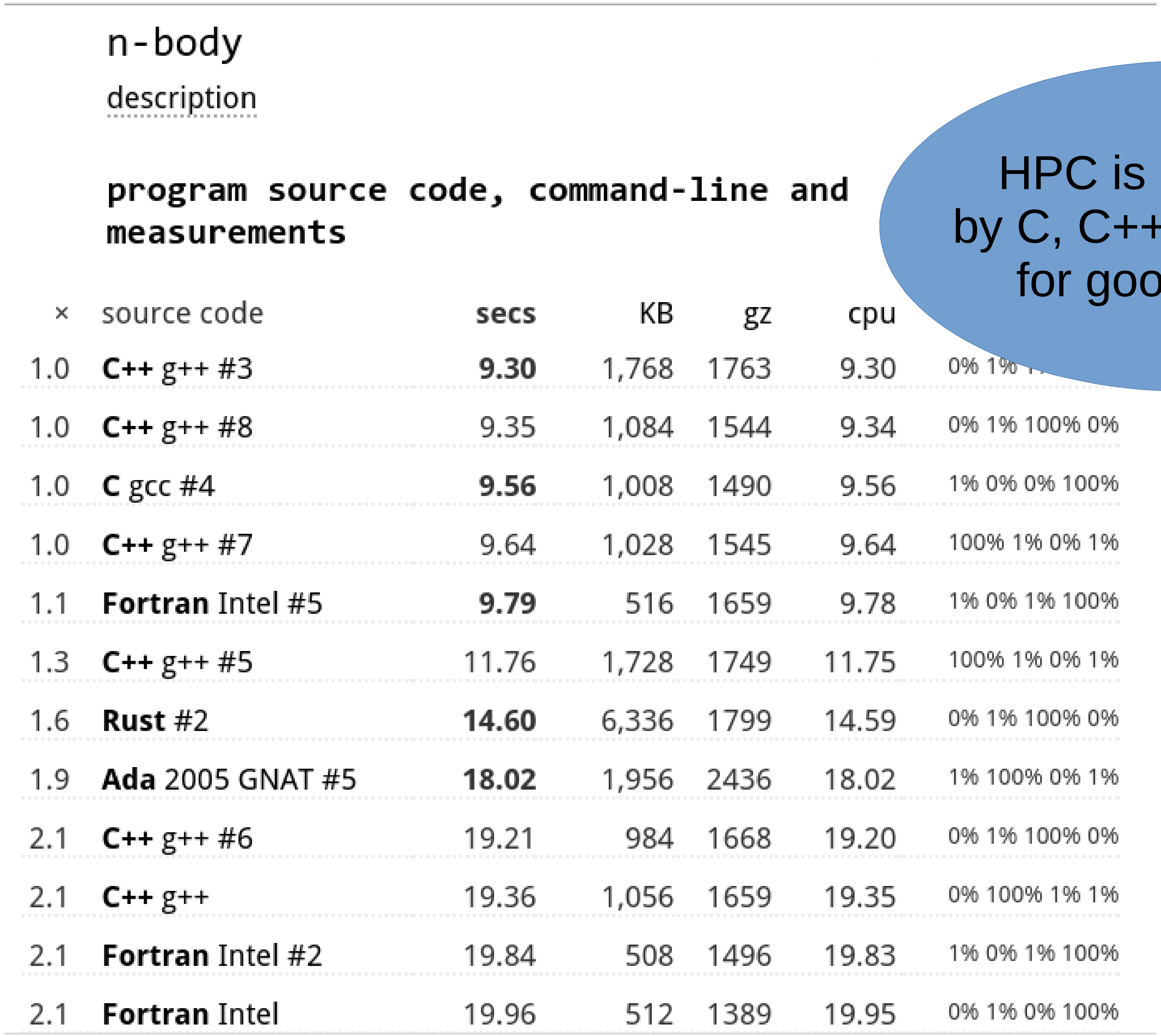
Performance Limited By...



HPC Languages

These top four all use vector intrinsics!

HPC is dominated by C, C++ and Fortran for good reason!



n-body

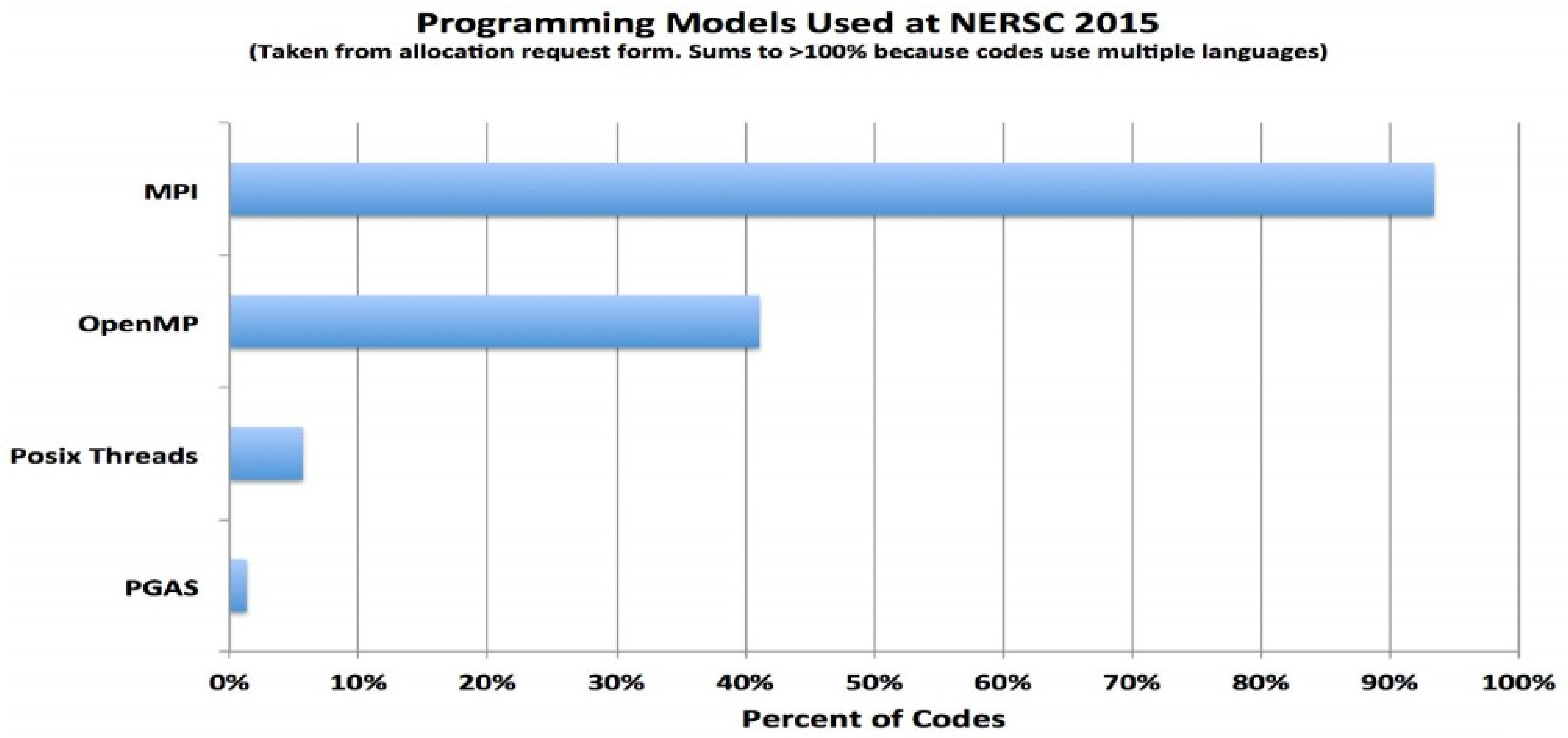
description

program source code, command-line and measurements

×	source code	secs	KB	gz	cpu	
1.0	**C++** g++ #3	**9.30**	1,768	1763	9.30	0% 1% 1
1.0	**C++** g++ #8	9.35	1,084	1544	9.34	0% 1% 100% 0%
1.0	**C** gcc #4	**9.56**	1,008	1490	9.56	1% 0% 0% 100%
1.0	**C++** g++ #7	9.64	1,028	1545	9.64	100% 1% 0% 1%
1.1	**Fortran** Intel #5	**9.79**	516	1659	9.78	1% 0% 1% 100%
1.3	**C++** g++ #5	11.76	1,728	1749	11.75	100% 1% 0% 1%
1.6	**Rust** #2	**14.60**	6,336	1799	14.59	0% 1% 100% 0%
1.9	**Ada** 2005 GNAT #5	**18.02**	1,956	2436	18.02	1% 100% 0% 1%
2.1	**C++** g++ #6	19.21	984	1668	19.20	0% 1% 100% 0%
2.1	**C++** g++	19.36	1,056	1659	19.35	0% 100% 1% 1%
2.1	**Fortran** Intel #2	19.84	508	1496	19.83	1% 0% 1% 100%
2.1	**Fortran** Intel	19.96	512	1389	19.95	0% 1% 0% 100%

https://benchmarksgame.alioth.debian.org/u64q/performance.php?test=nbody&sort=elapsed

How do we express parallelism?

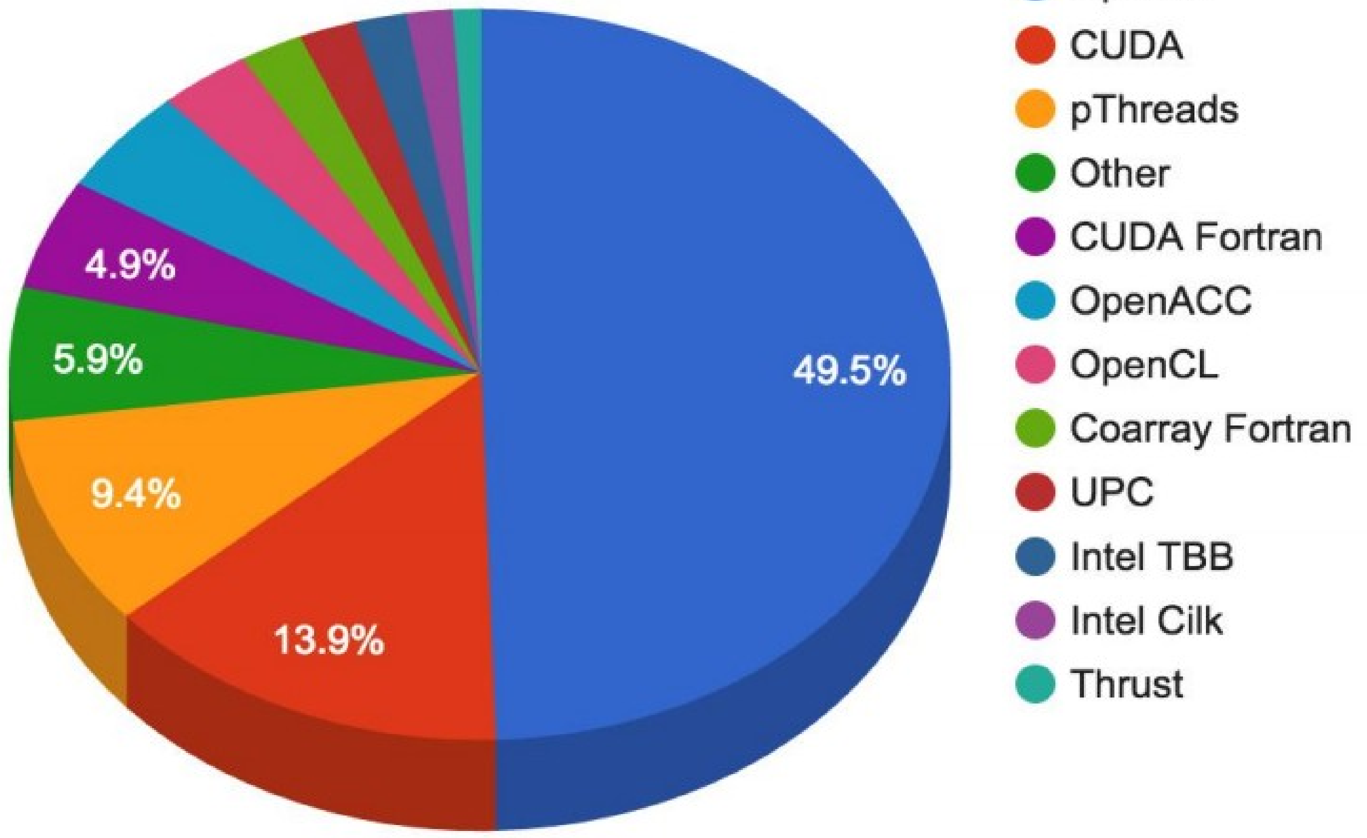


Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015

http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

How do we express parallelism - MPI+X?

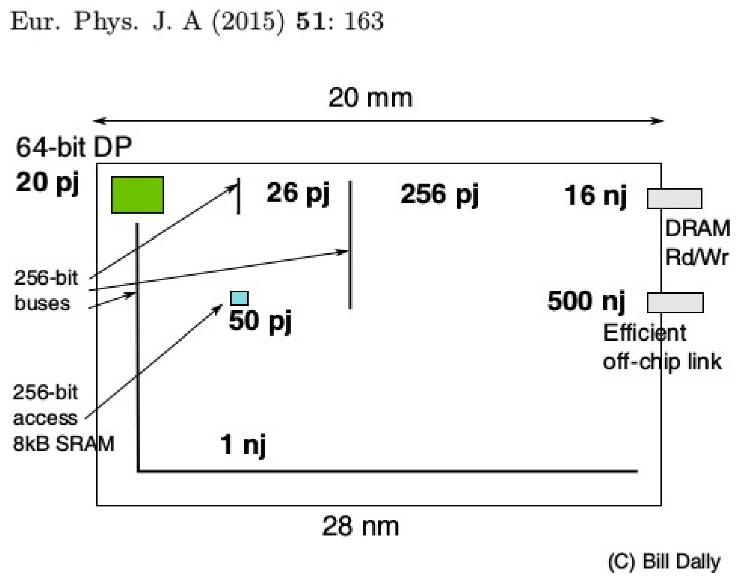
✓ OpenMP is about 50%, out of all choices of X

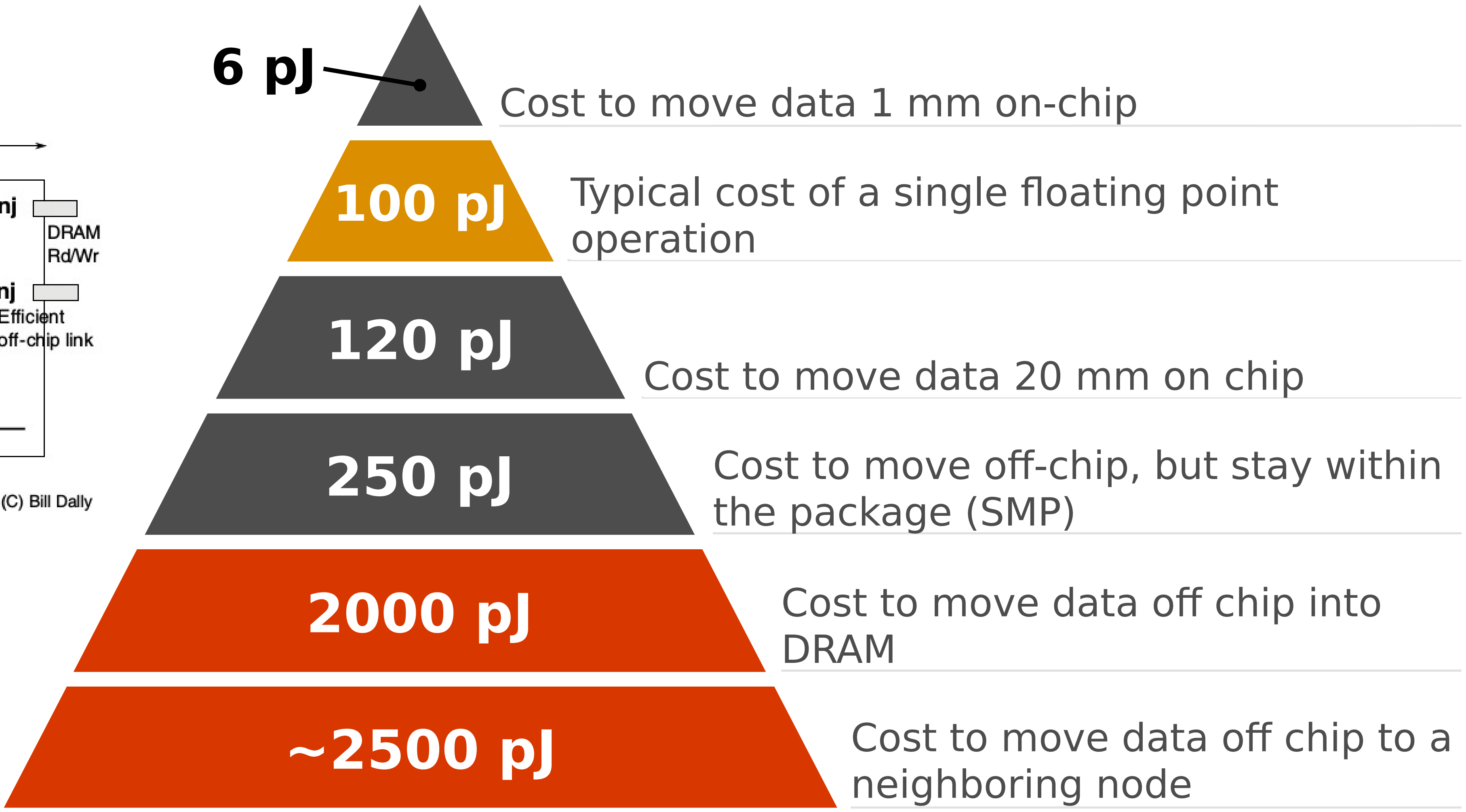


Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015

http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

The Challenge of the Future: Power



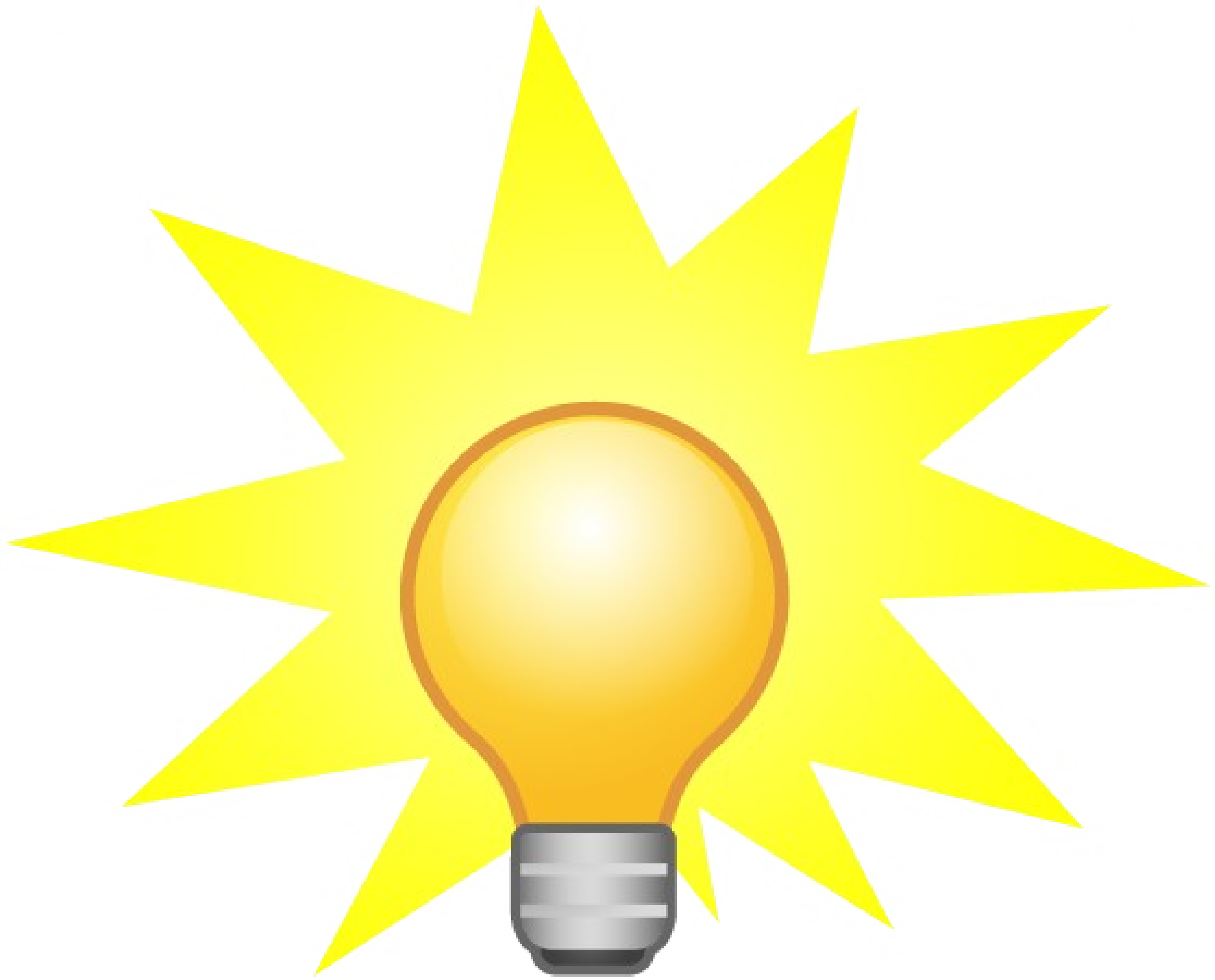

6 pJ
Cost to move data 1 mm on-chip
100 pJ
Typical cost of a single floating point operation
120 pJ
Cost to move data 20 mm on chip
250 pJ
Cost to move off-chip, but stay within the package (SMP)
2000 pJ
Cost to move data off chip into DRAM
~2500 pJ
Cost to move data off chip to a neighboring node

The DOE wants 1 exaflop at < 20 MW

http://www.socforhpc.org/wp-content/uploads/2015/07/OpenSoC_Pres_15min.pptx

About Using HPC Systems...

- Why is HPC Hard?
- What can you do about it?

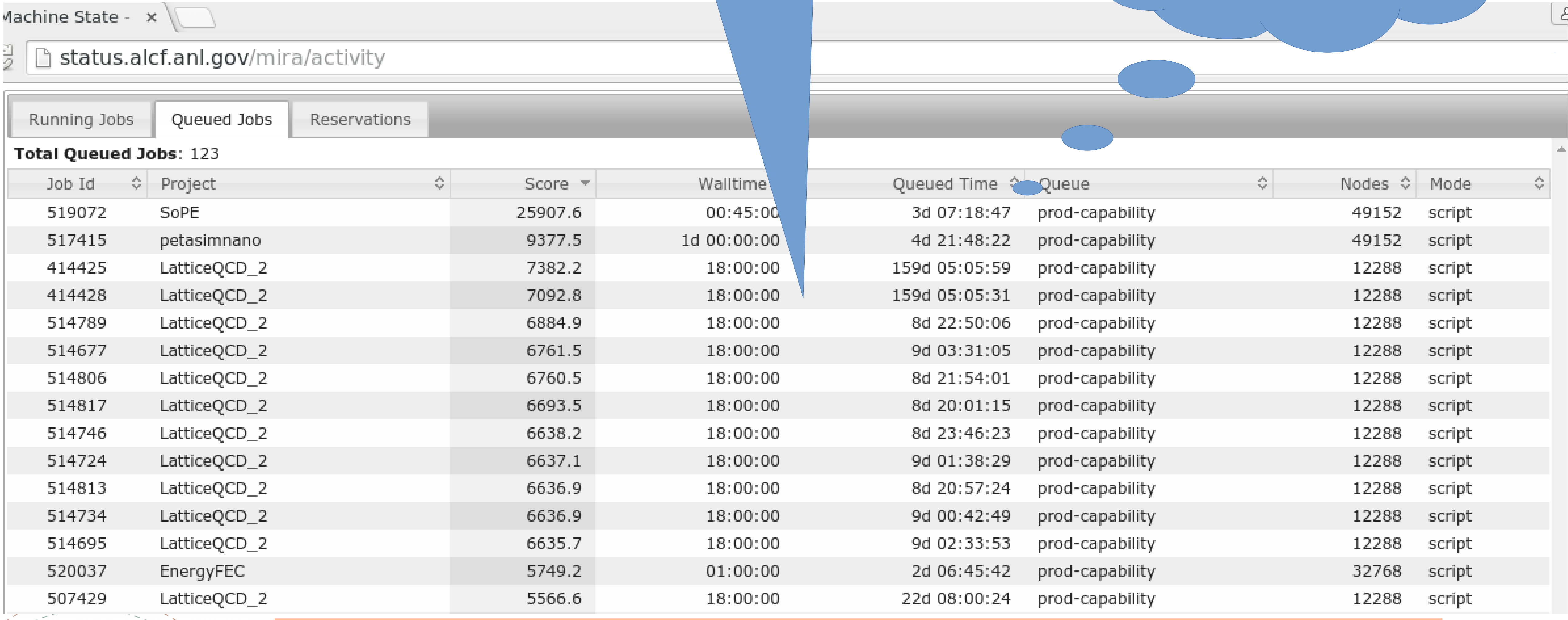


You need to share

Your job won't run right away...

These big jobs have been waiting for days

These were on hold for a while...
I hope



Machine State - ×

status.alcf.anl.gov/mira/activity

Running Jobs | Queued Jobs | Reservations

Total Queued Jobs: 123

Job Id	Project	Score	Walltime	Queued Time	Queue	Nodes	Mode
519072	SoPE	25907.6	00:45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	1d 00:00:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	18:00:00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8	18:00:00	159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9	18:00:00	8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	18:00:00	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	18:00:00	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18:00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

You have a limited amount of time

- A large allocation on Mira (ALCF's production resource) is a few hundred million core hours
- 100 M core hours / (16 (cores per node) * 1024 (nodes per rack) * 48 (racks)) == 127 hours
- 127 hours is 5.3 days
- Running on the whole machine (48 racks) for 24 hours is the largest possible job
- Thus, with 5 jobs (plus some test runs), a 100-M-core-hour allocation could be gone

Supercomputers are not commodity machines

- Even when built from commodity parts, the configuration and scale are different
- The probability that you'll try to do something in your application that has never been tested before is high
- The system software will have bugs, and the hardware might too.
- The libraries on which your code depends might not be available.

Your jobs will fail

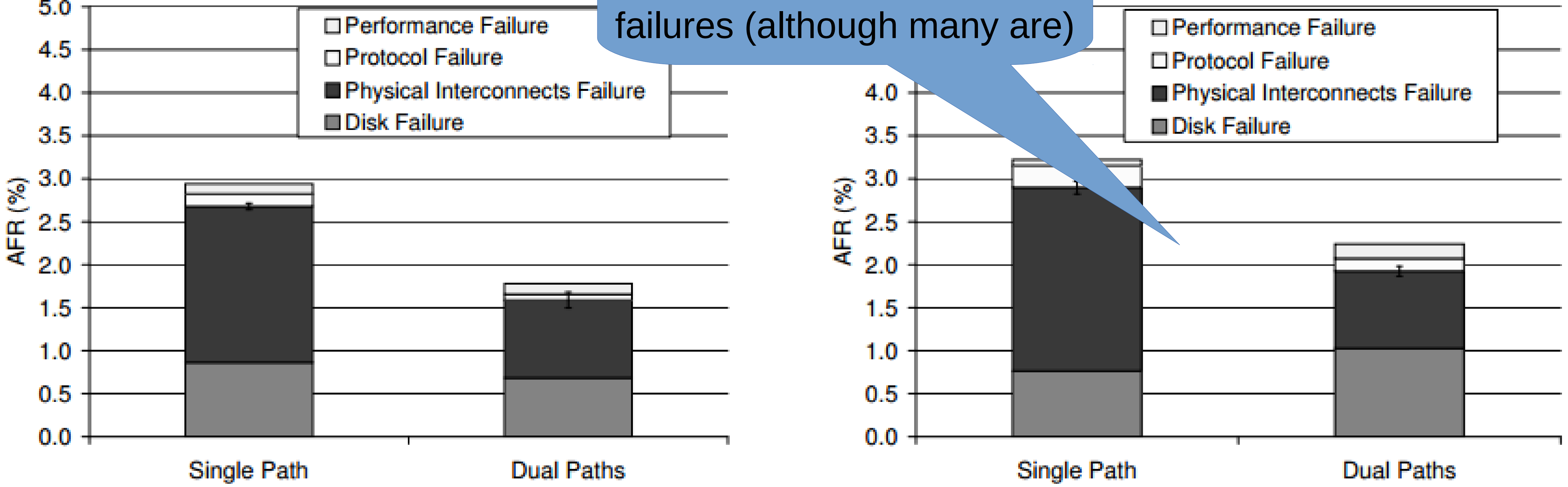
- The probability that a node will die, its DRAM will silently corrupt your data (including those in the storage subsystem), etc. is very low.
- However, if you spend a large fraction of your life running on large machines, you'll see these kinds of problems.

Your jobs will fail (cont.)

From the introduction of Fiala, et al. 2012:

- Servers tend to crash twice per year (2-4% failure rate) (Schroeder, et al. 2009). *HPC node hardware is designed to be somewhat more reliable, but...*
- 1-5% of disk drives die per year (Pinheiro, et al. 2007). *HPC storage systems use RAID, but... (also, many hardware RAID controllers don't do proper error checking, see Krioukov, et al. 2009 – older paper, but personal experience says this is still true today)*
- DRAM errors occur in 2% of all DIMMs per year (Schroeder, et al. 2009)
- ECC alone fails to detect a significant number of failures (Hwang, et al. 2012)

Your jobs will fail (cont.)



(a) Mid-range systems

(b) High-end systems

Most storage subsystem failures are not from disk failures (although many are)

Jiang, et al. 2008

Your jobs will fail (sometimes worse)

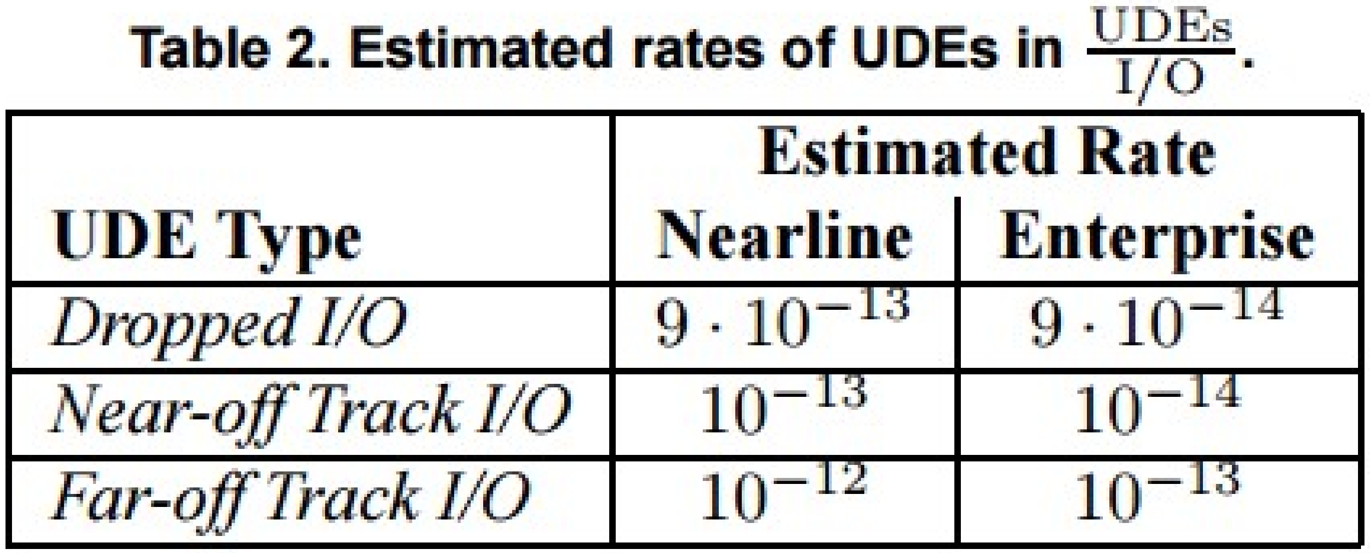


Table 2. Estimated rates of UDEs in $\frac{\text{UDEs}}{\text{I/O}}$.

UDE Type	Estimated Rate	
	Nearline	Enterprise
Dropped I/O	$9 \cdot 10^{-13}$	$9 \cdot 10^{-14}$
Near-off Track I/O	10^{-13}	10^{-14}
Far-off Track I/O	10^{-12}	10^{-13}

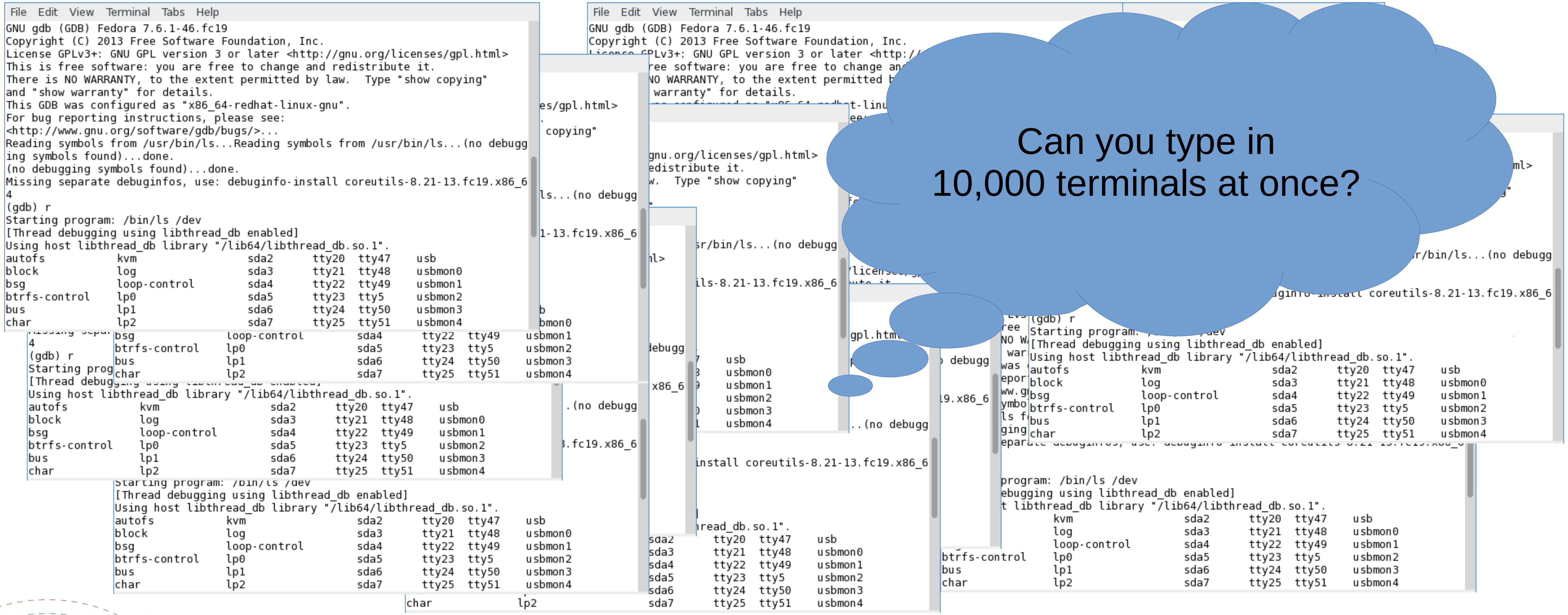
Rozier, et al. 2009 – UDE == Undetected Disk Error.
They assume 4 KB per I/O request

So you should expect to see silent data corruption once in every (even with RAID):
1/(2 x 10^-13) * 4*1024 == 2 x 10^16 bytes (20 PB)

Mira's file system is ~28 PB, so this is not an unthinkable number
(and, from personal experience, the rate is somewhat higher than that)

Debugging is hard

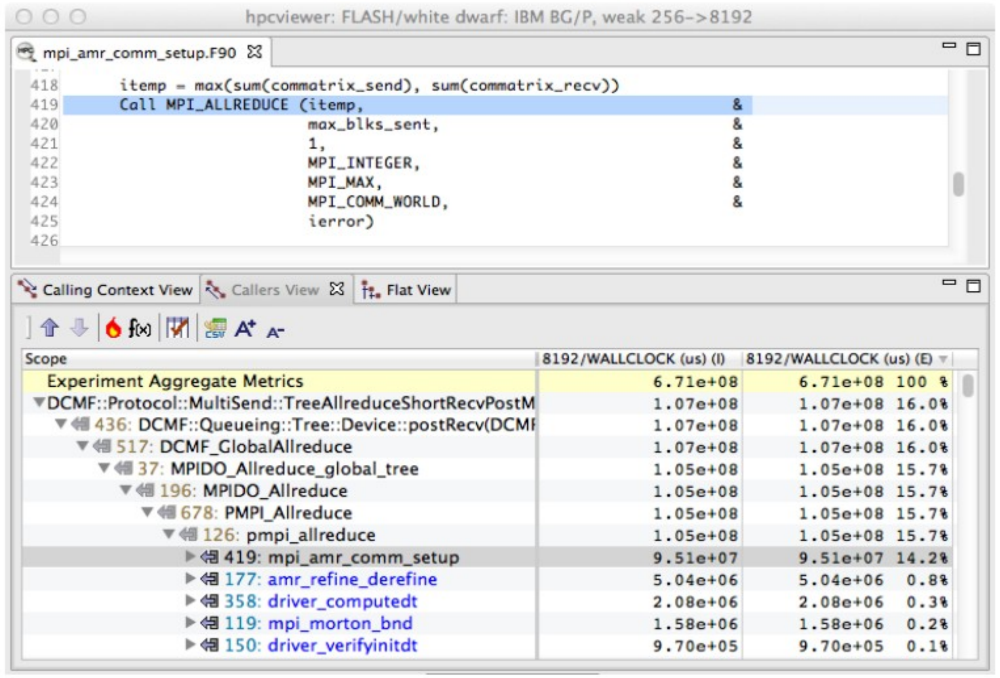
Your favorite debugger is great, but probably won't run at scale...


File Edit View Terminal Tabs Help
Can you type in 10,000 terminals at once?

```
GNU gdb (GDB) Fedora 7.6.1-46.fc19
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/bin/ls...Reading symbols from /usr/bin/ls...(no debugg
ing symbols found)...done.
(no debugging symbols found)...done.
Missing separate debuginfos, use: debuginfo-install coreutils-8.21-13.fc19.x86_6
4
(gdb) r
Starting program: /bin/ls /dev
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
autofs          kvm                  sda2      tty20  tty47    usb
block           log                  sda3      tty21  tty48    usbmon0
bsg             loop-control         sda4      tty22  tty49    usbmon1
btrfs-control   lp0                  sda5      tty23  tty5     usbmon2
bus             lp1                  sda6      tty24  tty50    usbmon3
char            lp2                  sda7      tty25  tty51    usbmon4
```

Debugging is hard (cont.)

Tools for debugging, profiling, etc. at scale are available, but they won't be what you're used to, and they might fail as well.



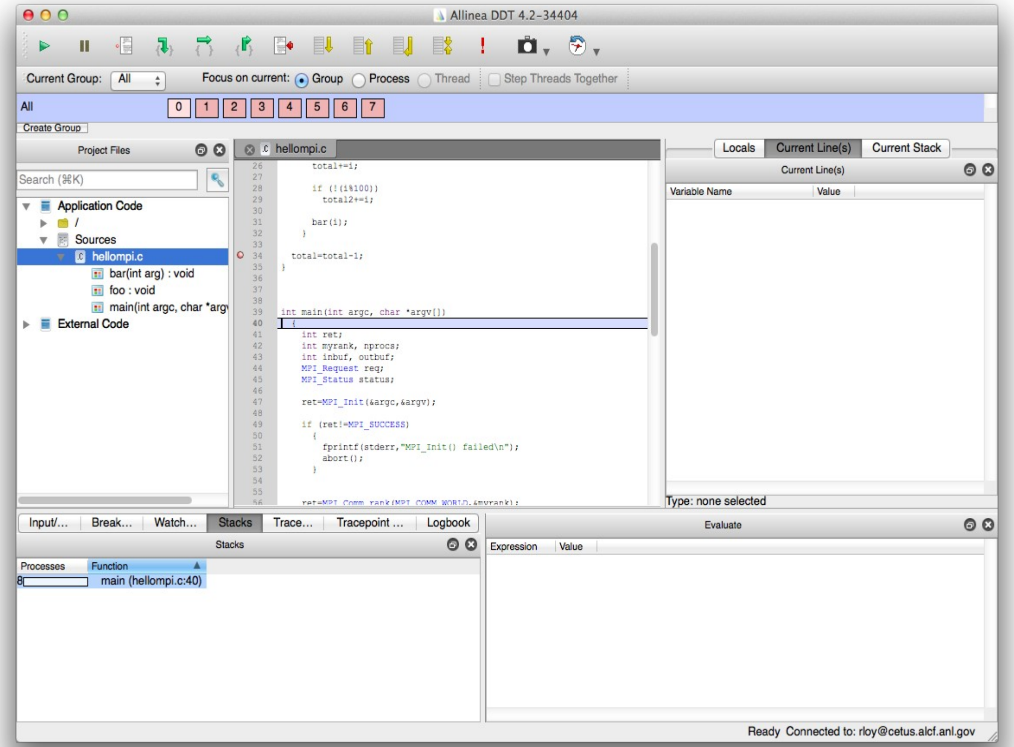
hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

mpi_amr_comm_setup.F90

```
      itemp = max(sum(commatrix_send), sum(commatrix_recv))
      Call MPI_ALLREDUCE (itemp,                              &
                          max_blks_sent,                      &
                          1,                                  &
                          MPI_INTEGER,                        &
                          MPI_MAX,                            &
                          MPI_COMM_WORLD,                     &
                          ierror)
```

```
Scope                                              8192/WALLCLOCK (us) (I)  8192/WALLCLOCK (us) (E)
 Experiment Aggregate Metrics                                   6.71e+08    6.71e+08 100 %
▼DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostM         1.07e+08    1.07e+08 16.0%
  ▼ 436: DCMF::Queueing::Tree::Device::postRecv(DCMF            1.07e+08    1.07e+08 16.0%
    ▼ 517: DCMF_GlobalAllreduce                                 1.07e+08    1.07e+08 16.0%
      ▼ 37: MPIDO_Allreduce_global_tree                         1.05e+08    1.05e+08 15.7%
        ▼ 196: MPIDO_Allreduce                                  1.05e+08    1.05e+08 15.7%
          ▼ 678: PMPI_Allreduce                                 1.05e+08    1.05e+08 15.7%
            ▼ 126: pmpi_allreduce                               1.05e+08    1.05e+08 15.7%
              ▶ 419: mpi_amr_comm_setup                         9.51e+07    9.51e+07 14.2%
              ▶ 177: amr_refine_derefine                        5.04e+06    5.04e+06  0.8%
              ▶ 358: driver_computedt                           2.08e+06    2.08e+06  0.3%
              ▶ 119: mpi_morton_bnd                             1.58e+06    1.58e+06  0.2%
              ▶ 150: driver_verifyinitdt                        9.70e+05    9.70e+05  0.1%
```

HPCToolkit



Allinea DDT 4.2-34404

Current Group: All　Focus on current: (● Group) Process Thread　Step Threads Together

hellompi.c

```
      total+=i;

      if (!(i%100))
        total2+=i;

      bar(i);
    }

  total=total-1;
}



int main(int argc, char *argv[])
  {
    int ret;
    int myrank, nprocs;
    int inbuf, outbuf;
    MPI_Request req;
    MPI_Status status;

    ret=MPI_Init(&argc,&argv);

    if (ret!=MPI_SUCCESS)
      {
        fprintf(stderr,"MPI_Init() failed\n");
        abort();
      }


    ret=MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

Stacks: 8 main (hellompi.c:40)

Ready Connected to: rloy@cetus.alcf.anl.gov

Allinea DDT

Everything is fast, but too slow...

- The network is fast, but likely slower than you'd like
- The same is true for the memory subsystem
- The same is true for the storage subsystem

Mira's file system can provide 240 GB/s – but you need to use the whole machine to get that rate. In addition, writing is slower than reading.

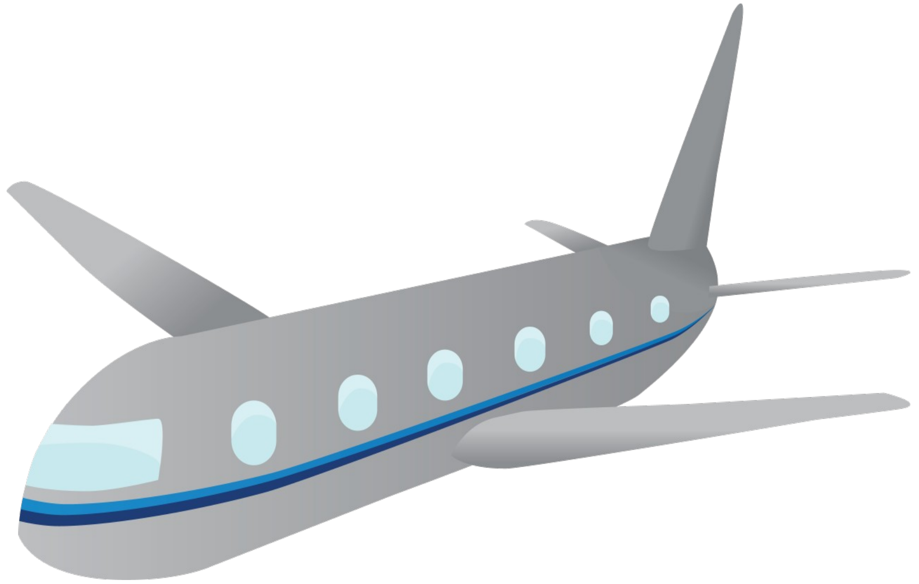
If you had the whole machine, and were the only one using the file system, then reading in enough to fill all 768 TB of DRAM would take:

(768*1024 / 240)/60 == 55 minutes

Everything is fast, but too slow... (cont.)

Also remember that:

- There is a big difference between latency and bandwidth
- I/O performance tends to be quirky (memory performance does too, to a lesser extent)
- Load balancing can be tricky

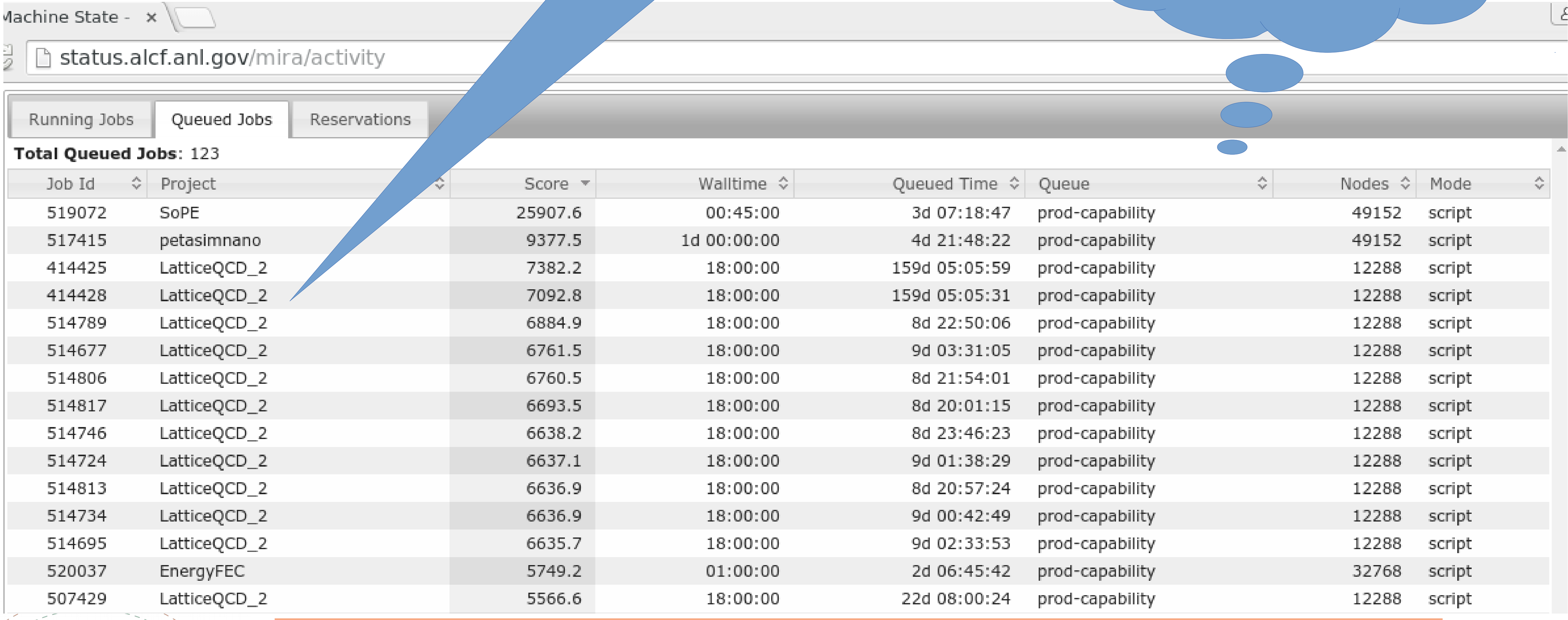


So, now what do you do?

Experiment in parallel

This guy has the right idea

Some of these (time,size) combinations are better than others

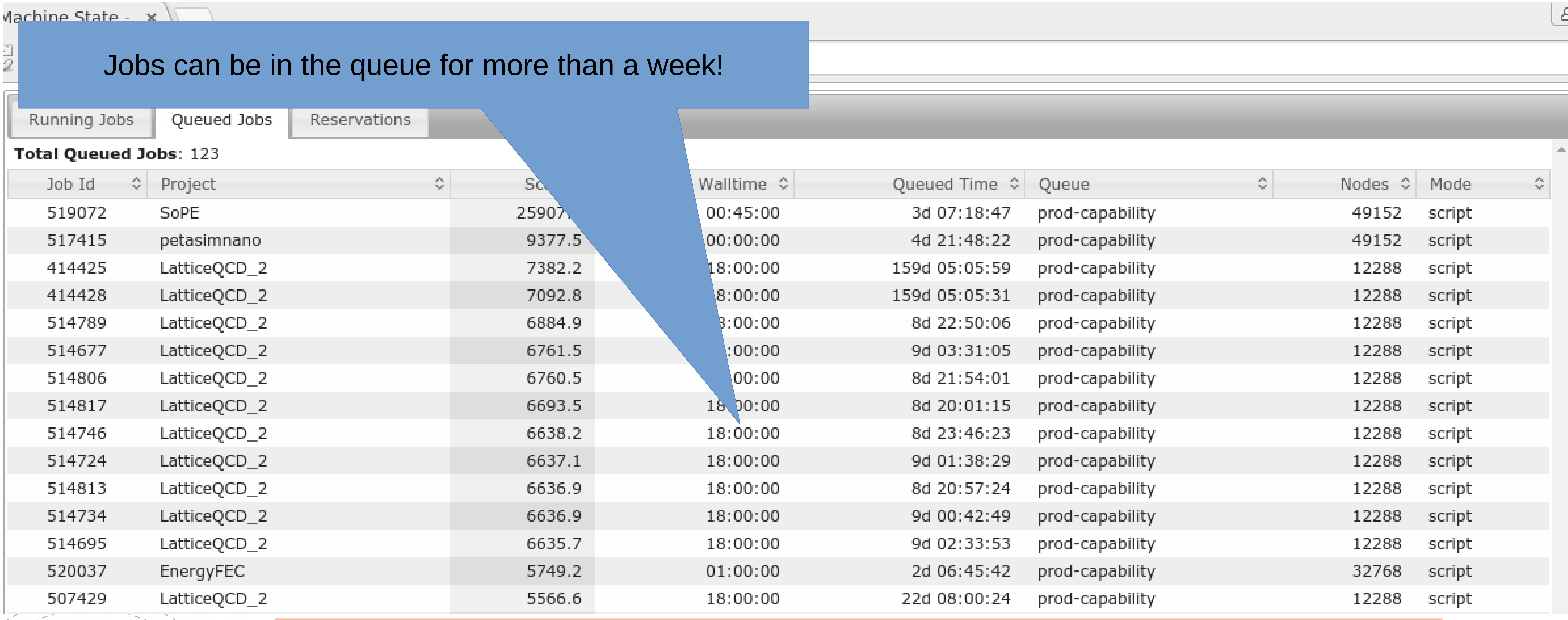


Total Queued Jobs: 123

Job Id	Project	Score	Walltime	Queued Time	Queue	Nodes	Mode
519072	SoPE	25907.6	00:45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	1d 00:00:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	18:00:00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8	18:00:00	159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9	18:00:00	8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	18:00:00	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	18:00:00	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18:00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

Don't wait until the last minute...

Your allocation probably ends along with many others, and many users procrastinate, don't be one of them!



Jobs can be in the queue for more than a week!

Total Queued Jobs: 123

Job Id	Project	Sc	Walltime	Queued Time	Queue	Nodes	Mode
519072	SoPE	25907	00:45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	00:00:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	18:00:00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8	8:00:00	159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9	3:00:00	8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	:00:00	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	00:00	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18 00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

Make your code exit!

“Begin at the beginning,” the King said, very gravely, “and go on till you come to the end: then stop.”
- Lewis Carroll, Alice in Wonderland

On Mira, 1 minute on the whole machine is 13,107 core hours!

- Take the time to figure out how long your code takes to run, and make it exit (don't always run your jobs until the system kills them).
- Exiting cleanly (using an exit code of 0), is often necessary for dependency chaining to work.

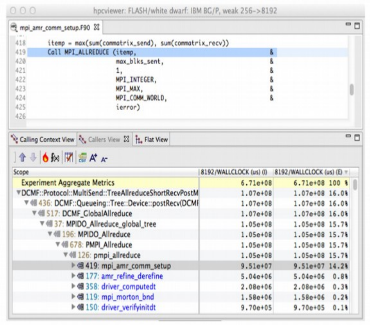
Save your work

- Save all of your configuration files for any experiments you do.
- Save your log files.
- Document how to run your code and process the results, what you've actually run, and where the data lives.

Store the run configuration in every output file
(as comments, metadata, etc.)
Seriously, just do it...

Optimize your code

- Read some documentation on the system, compiler, etc.
- Do you need strict IEEE floating point semantics? If not, turn them off.
- Profile your code
- Then optimize (using better algorithms first)



```
      itemp = max(sum(commatrix_send), sum(commatrix_recv))
      Call MPI_ALLREDUCE (itemp,                                   &
                          max_blks_sent,                           &
                          1,                                       &
                          MPI_INTEGER,                             &
                          MPI_MAX,                                 &
                          MPI_COMM_WORLD,                          &
                          ierror)
```

Scope	8192/WALLCLOCK (us) (I)	8192/WALLCLOCK (us) (E)
Experiment Aggregate Metrics	6.71e+08	6.71e+08 100 %
DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostM	1.07e+08	1.07e+08 16.0%
436: DCMF::Queueing::Tree::Device::postRecv(DCMF	1.07e+08	1.07e+08 16.0%
517: DCMF_GlobalAllreduce	1.07e+08	1.07e+08 16.0%
37: MPIDO_Allreduce_global_tree	1.05e+08	1.05e+08 15.7%
196: MPIDO_Allreduce	1.05e+08	1.05e+08 15.7%
678: PMPI_Allreduce	1.05e+08	1.05e+08 15.7%
126: pmpi_allreduce	1.05e+08	1.05e+08 15.7%
419: mpi_amr_comm_setup	9.51e+07	9.51e+07 14.2%
177: amr_refine_derefine	5.04e+06	5.04e+06 0.8%
358: driver_computedt	2.08e+06	2.08e+06 0.3%
119: mpi_morton_bnd	1.58e+06	1.58e+06 0.2%
150: driver_verifyinitdt	9.70e+05	9.70e+05 0.1%

Always compile with -g (debugging symbols)
Essential for debugging if your code crashes,
but also useful for profiling, etc.

Contact support

If something is wrong, or you need help for any other reason, contact the facility's support service:

(system reservations for debugging are often possible, just ask!)

These people are not scary!

On using libraries

Using libraries written by experts is really important, but remember that if you're using something obscure, you'll “own” that dependency.

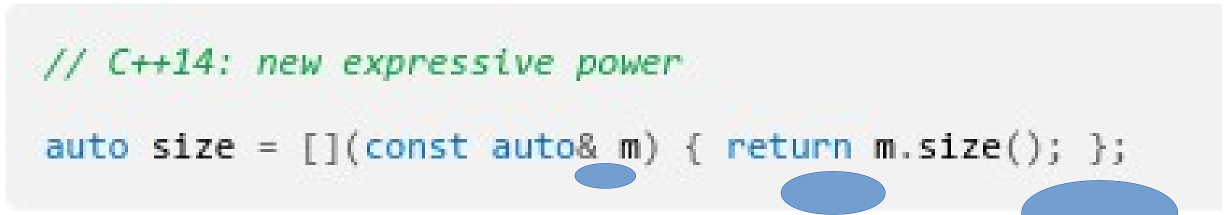
https://en.wikipedia.org/wiki/File:Two_Paths_Diverged_in_a_wood.JPG

Some popular libraries:

- ✓ Trilinos
- ✓ PETSc
- ✓ HDF5
- ✓ FFTW
- ✓ BLAS/LAPACK

Programming-Language Features

- Be careful when using the latest-and-greatest programming-language features
- We're just getting C++11 and Fortran 2008 support in compilers now (not counting co-arrays)



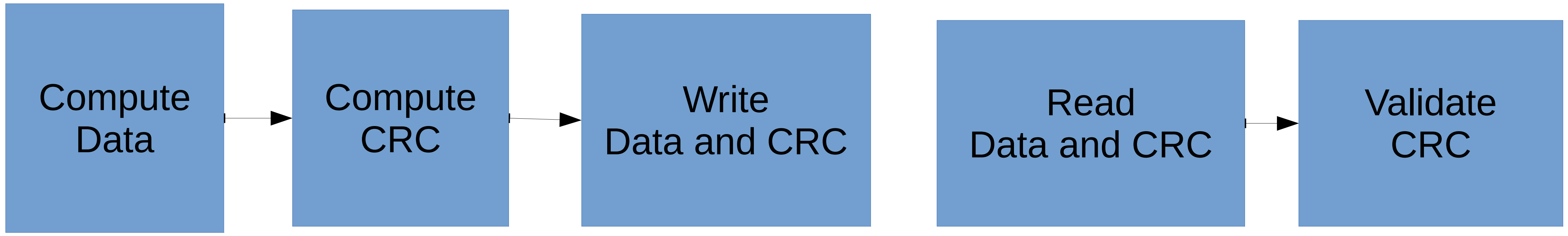
```
// C++14: new expressive power
auto size = [](const auto& m) { return m.size(); };
```

Yeah, not so fast...

(the facilities try their best to support these things using open-source compilers, etc. but, as a user, you'll likely want the option of using the vendor's compilers)

Validation

- Have test problems, hopefully both small and large ones, and run them on the system, with the same binary you plan to use for production, before starting your production science.
- Make sure all of the data files have checksums (CRCs) so that you can validate that the data you wrote is the same as the data you read in.
- Build physical diagnostics into your simulations (conservation of energy, power spectrum calculations, etc.) and actually check them.



Save your build settings

- Make your code print out or save its configuration when it starts, and also:
  - The compilers and build flags used
  - The version control revision information for the source being built

If you don't know what version control is, learn about git.

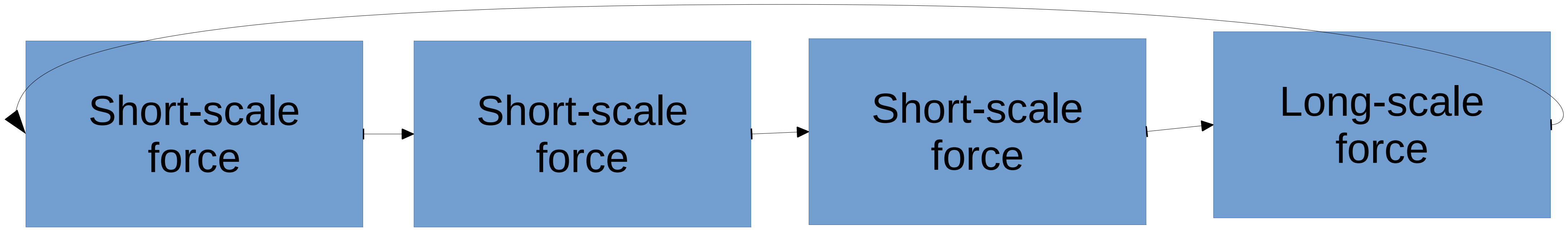
There is no general automatic way to do these things: You'll need to hack your build system. It will be worth it.

Some thoughts on testing

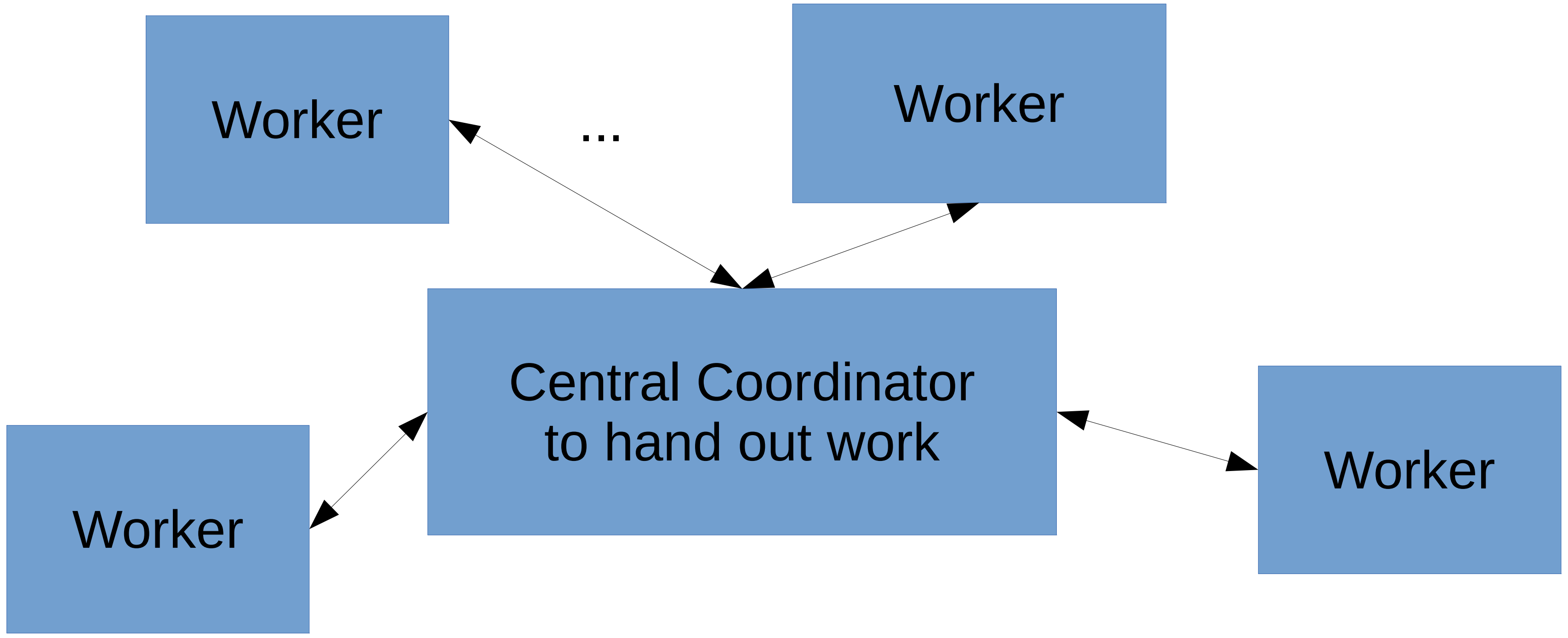
- Make as much of your code testable at small scale as possible.
- Unit testing is trendy for a good reason.
- Learn how to use Valgrind, and run your code at small scale with it.
- Add print statements in your code for anomalous situations: lots of them.
- Make sure you actually check for error return codes on routines that have them (for MPI, communication failures will kill your application by default, file I/O errors won't't).

Avoid the network

- Network bandwidth, relative to FLOPS, is decreasing
- Choose, to the extent possible, communication-avoiding algorithms
- If your problem has multiple physical time/length scales, try to separate out the shorter/faster ones and keep them rank-local (local sub-cycling).
- More generally, learn about split-operator methods.



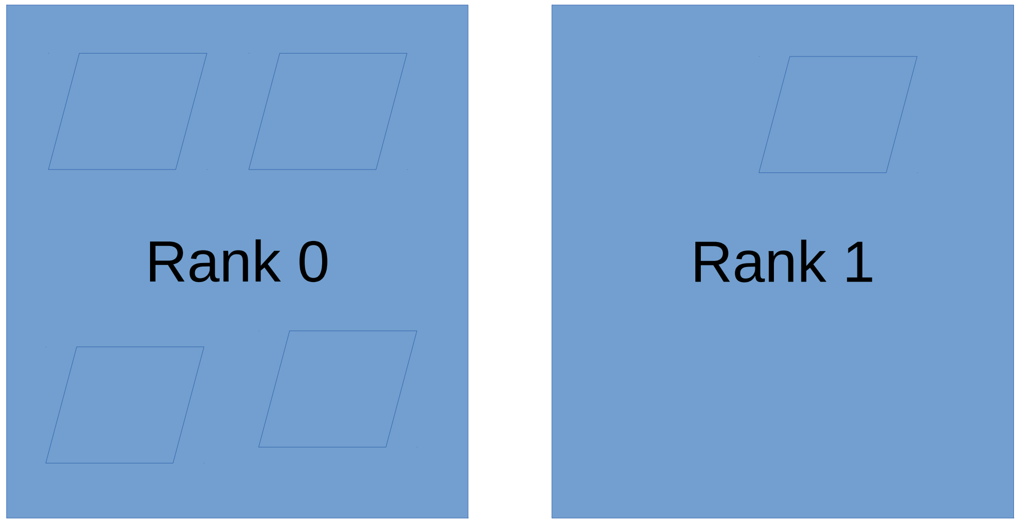
Avoid central coordinators



A scheme like this is highly unlikely to scale!

Load Balancing

- Keep "work units" being distributed between ranks as large as possible, but try hard to keep everything load balanced.
- Think about load balancing early in your application design: it is the largest impediment to scaling on large systems.



This is not good; rank 0 has much more work.

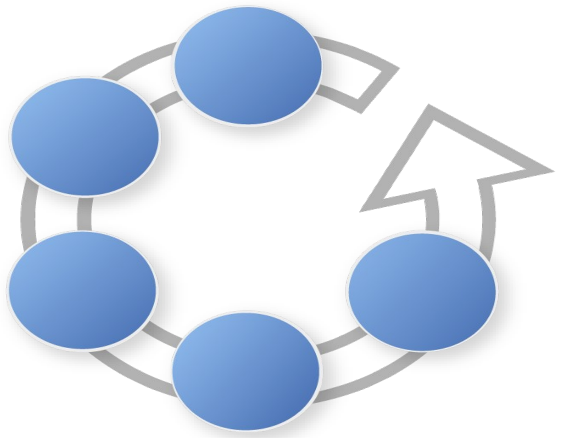
Memory Bandwidth

- Memory bandwidth will be low compared to the compute capability of the machine.
- To get the most out of the machine, you'll need to use FMA instructions (these machines were built to evaluate polynomials, many of them in parallel, so try to cast what you're doing in those terms).
- Try to do as much as possible with every data value you load, and remember that gathering data from all over memory is expensive.

FMA = Fused Multiply Add = (a * b) + c [with no intermediate rounding]

Expensive Algorithms

- Don't dismiss seemingly-expensive algorithms without benchmarking them (higher-order solvers, forward uncertainty propagation, etc. all might have high data reuse so the extra computational expense might be “free”).
- If you're using an iterative solver, the number of iterations you use will often dominate over the expense of each iteration.



More on I/O

This is a pain,
I know.

- Make your files “write optimized”.
- Don't use one file per rank, but don't have all ranks necessarily write to the same file either: make the number of files configurable.
- The optimal mapping between ranks and files will be system specific (ask the system experts what this is).
- There is often lock contention on blocks, files, directories, etc.
- Pre-allocate your file extents when possible.
- Use collective MPI I/O when the amount of data per rank is small (a few MB or less per rank).

Some final advice...

Don't guess! Profile! Your performance bottlenecks might be very different on different systems.

And don't be afraid to ask questions...

?

Any questions?

Come to the focus session tonight!