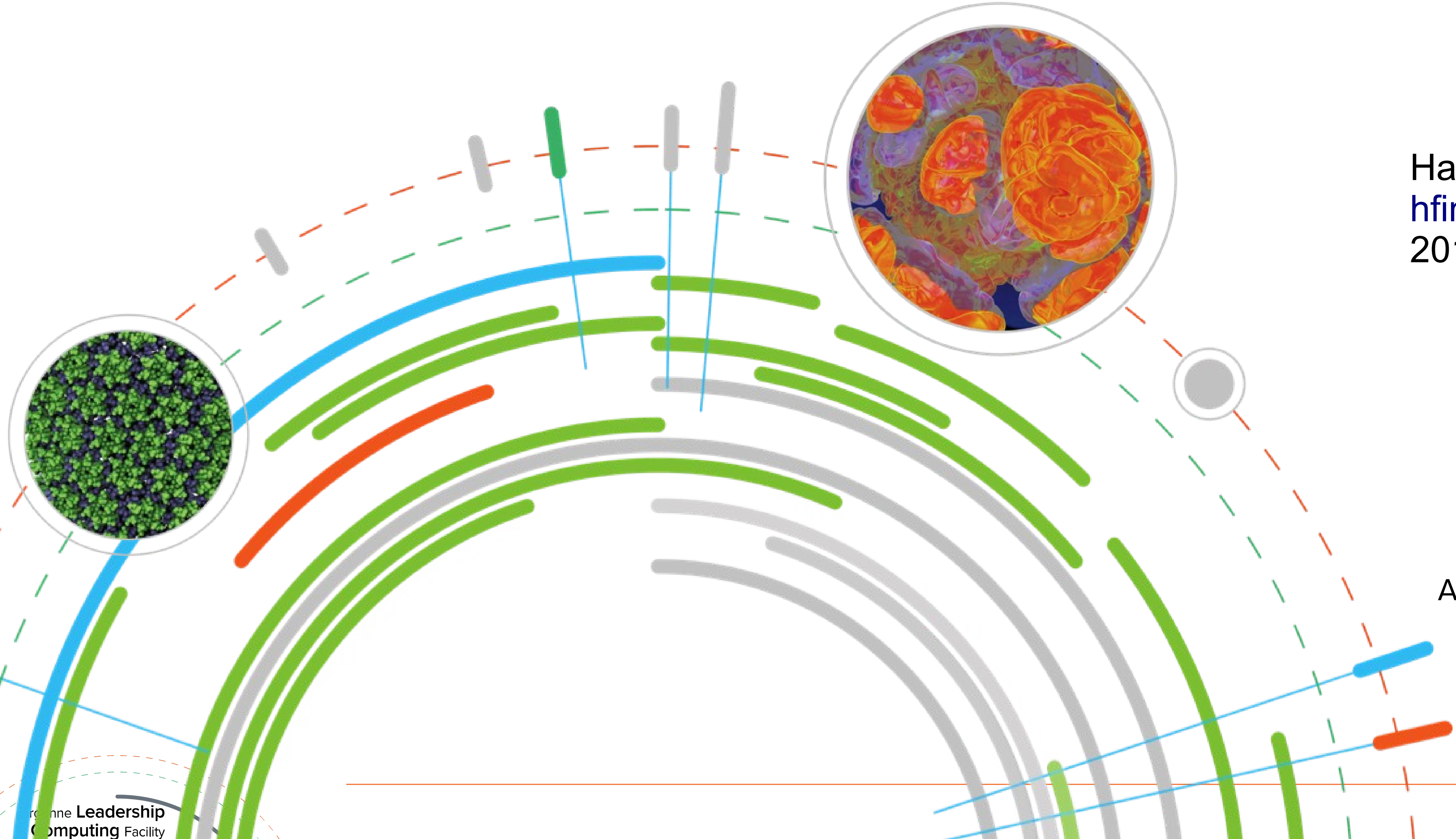
Optimizing for Blue Gene/Q

Hal Finkel
hfinkel@anl.gov
2015-05-19



Argonne Leadership Computing Facility

Argonne NATIONAL LABORATORY

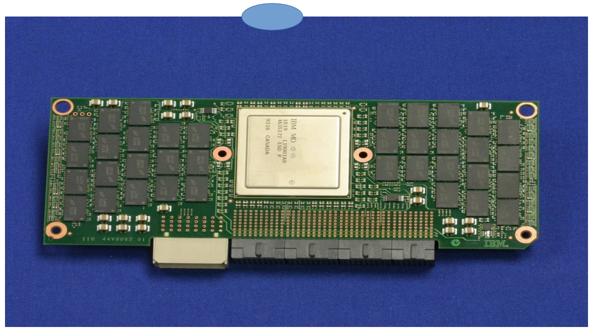
Optimizing for Blue Gene/Q

You want to know how
to make me compute quickly...

- ✓ Relevant information on the BG/Q
- ✓ How you can optimize your code for the BG/Q
- ✓ Q&A

Optimizing for Blue Gene/Q

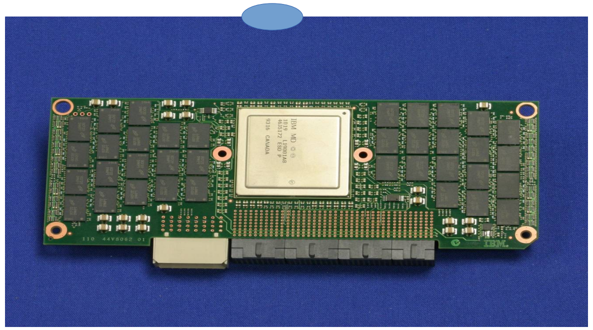
This is a BG/Q node



This is not

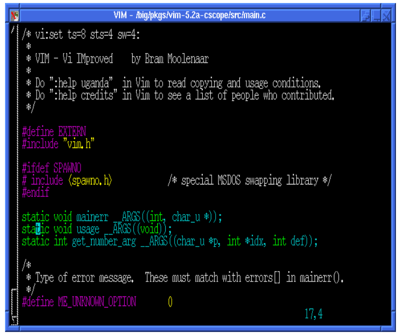
Optimizing for Blue Gene/Q

This is a BG/Q node



Mira has 49152 of these functioning as compute nodes!

What programs do...



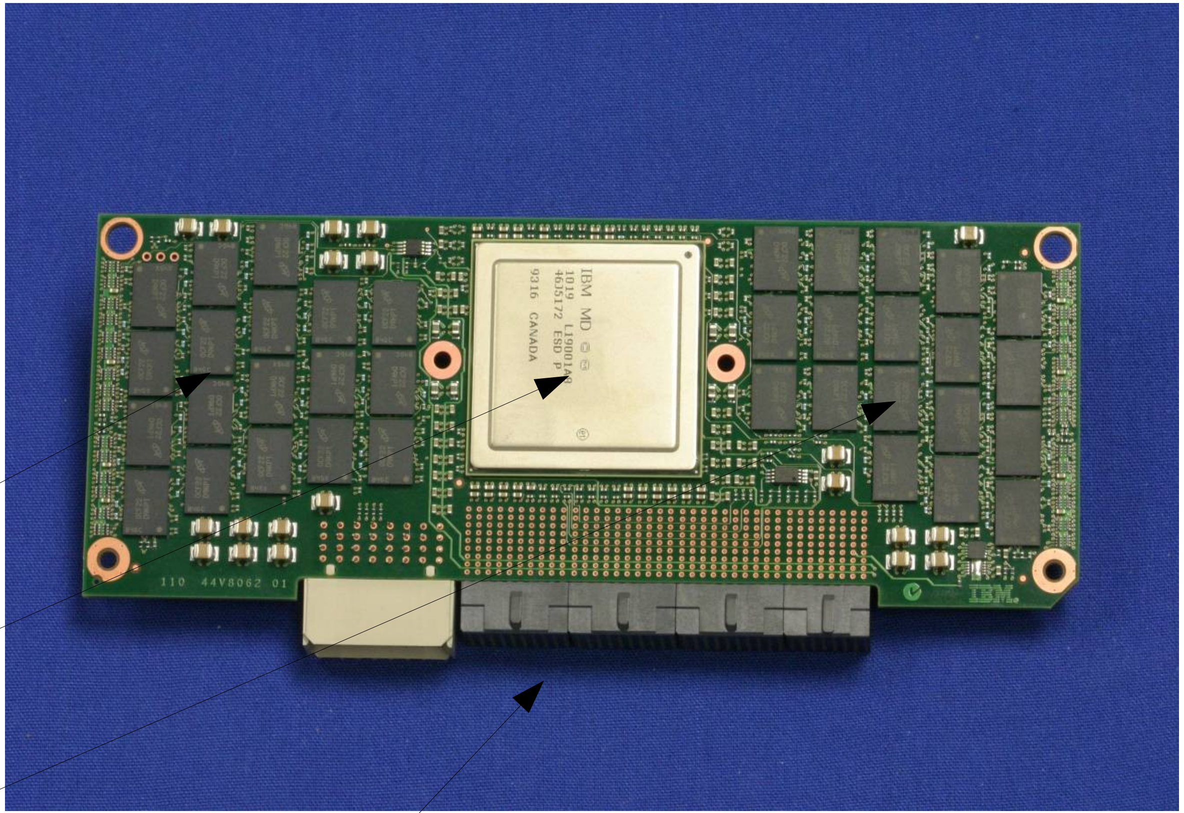
```
/* vi:set ts=8 sts=4 sw=4:
 *
 * VIM - Vi IMproved    by Bram Moolenaar
 *
 * Do ":help uganda"  in Vim to read copying and usage conditions.
 * Do ":help credits" in Vim to see a list of people who contributed.
 */

#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

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)

This is fixed:
1.66 GHz

Your hard work goes here...

Types of parallelism

- Parallelism across nodes (using MPI, etc.)
- Parallelism across sockets within a node [Not applicable to the BG/Q]
- 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!

There is only one socket

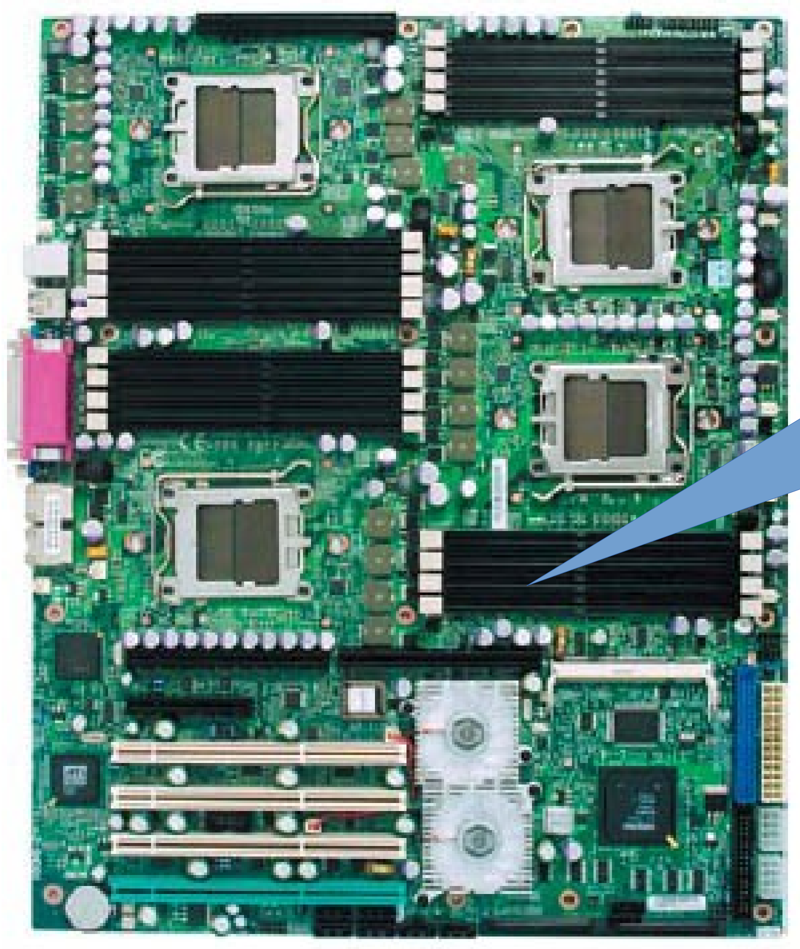
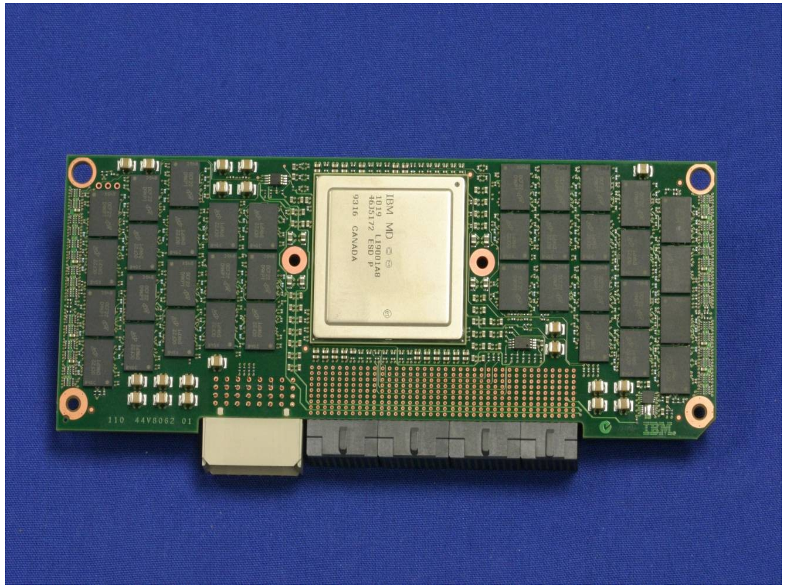


Image source: https://computing.llnl.gov/tutorials/linux_clusters/

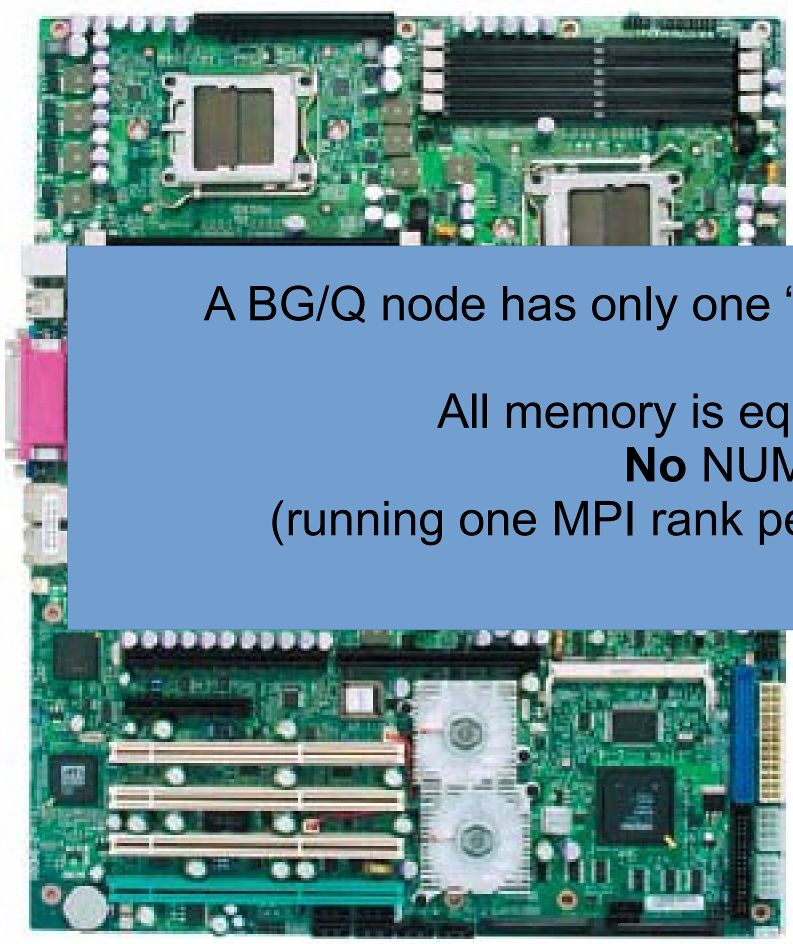
Commodity HPC node with four sockets

Has nonuniform memory access (NUMA):
each core has DRAM to which it is closer
(running multiple MPI ranks per node, one per socket, is probably best)

Not a BG/Q node



There is only one socket

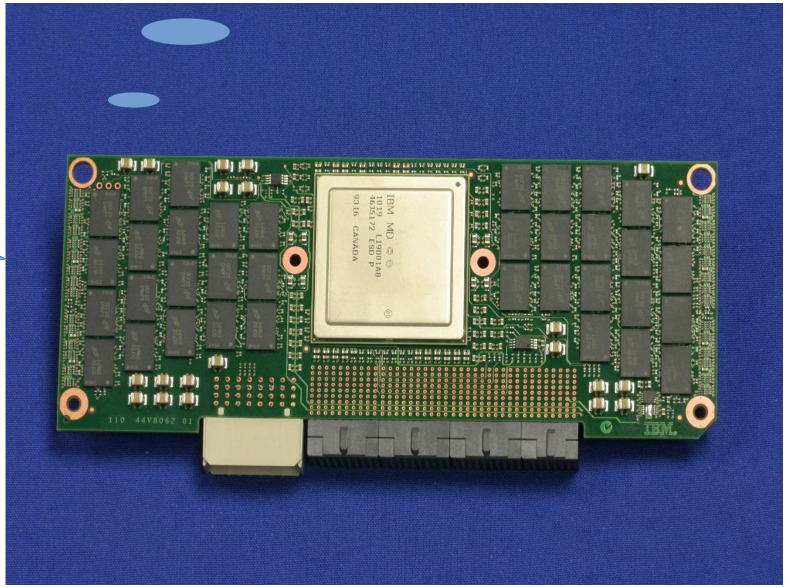


A BG/Q node has only one “socket” with one CPU

All memory is equally close:
No NUMA
(running one MPI rank per node works well)

Image source: https://computing.llnl.gov/tutorials/linux_clusters/

A BG/Q node



A BG/Q Node has:

- ✓ 1 PowerPC A2Q CPU
- ✓ 16 GB DDR3 DRAM

There are 16 cores per node

Not a BG/Q core

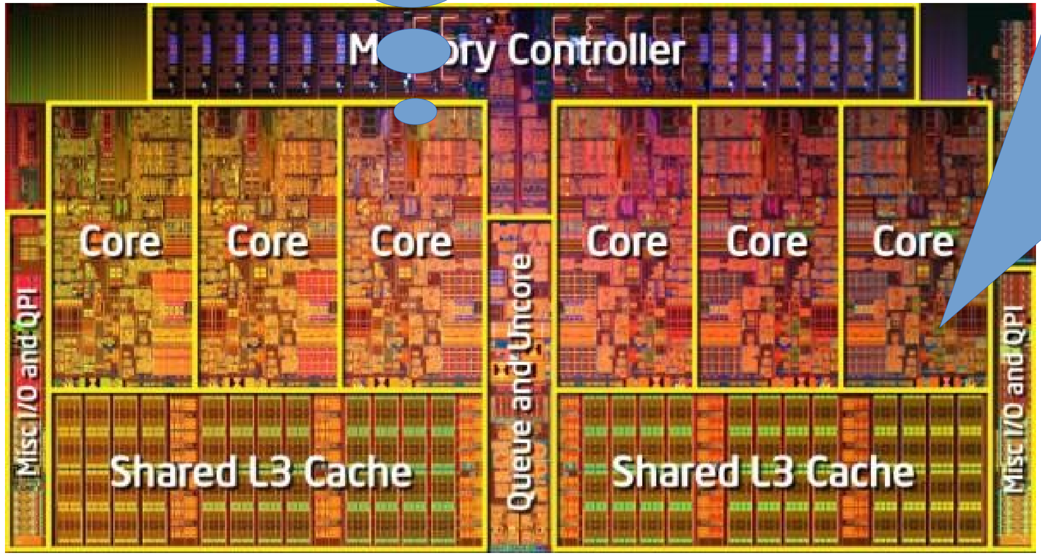
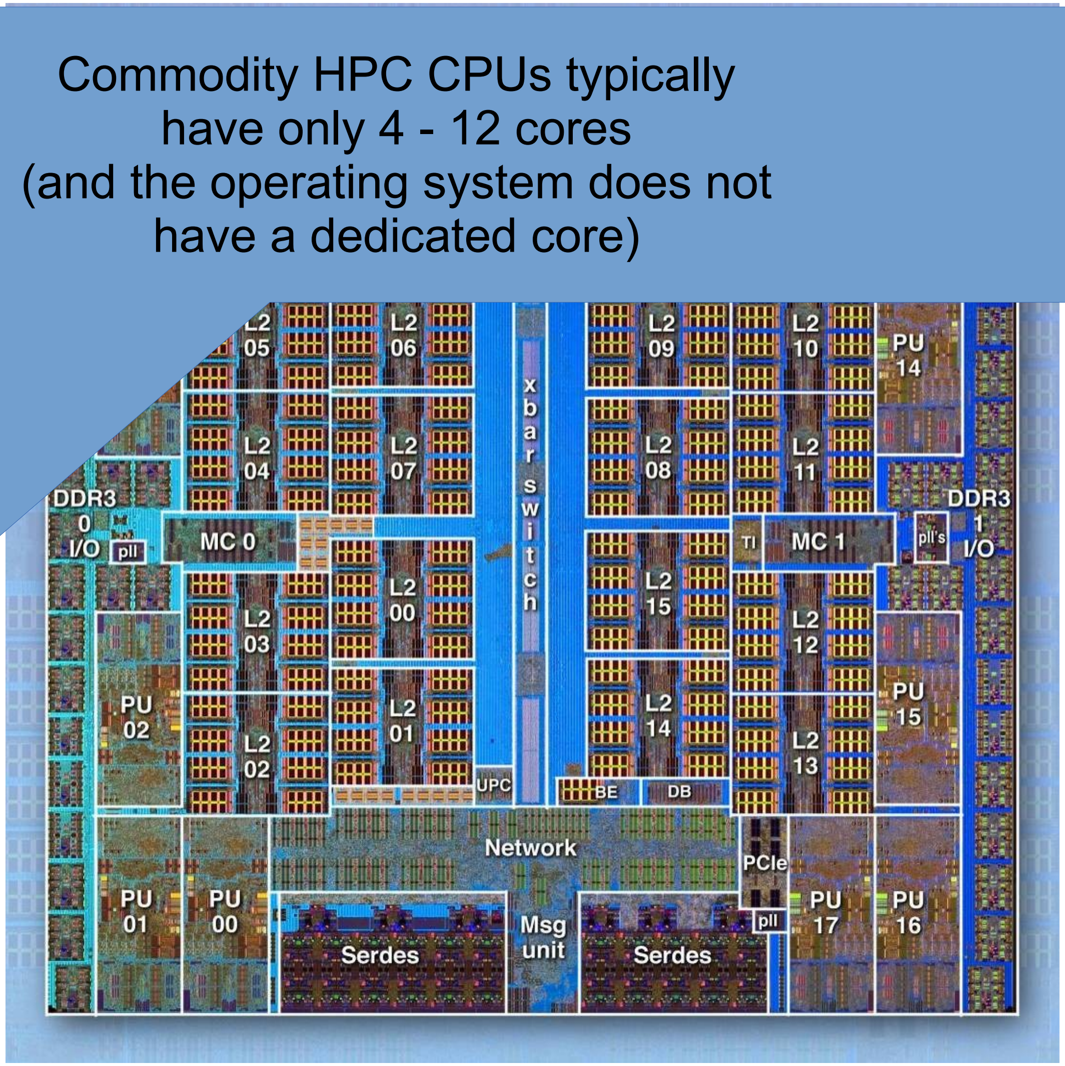


Image source: https://computing.llnl.gov/tutorials/linux_clusters/

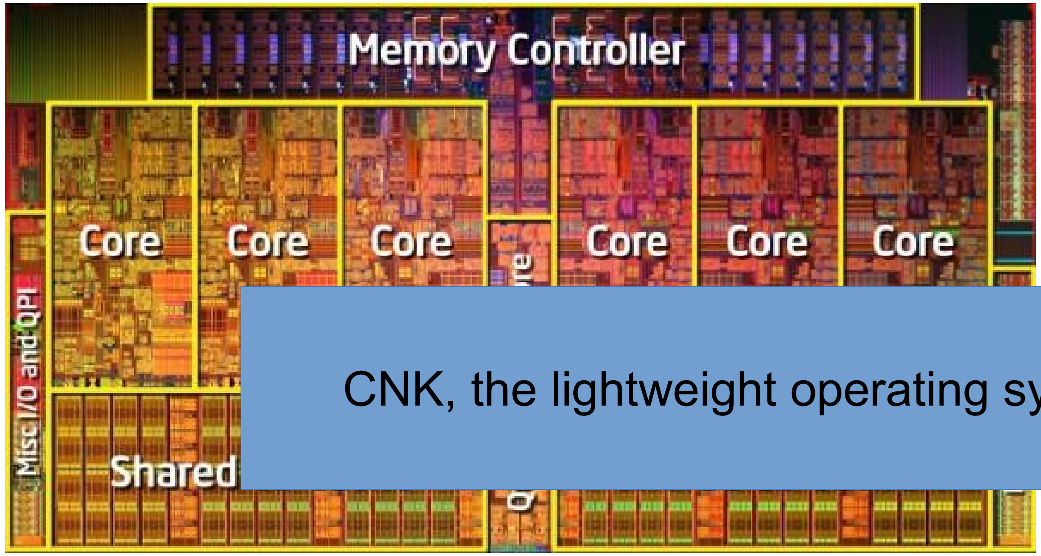
Commodity HPC CPUs typically have only 4 - 12 cores (and the operating system does not have a dedicated core)

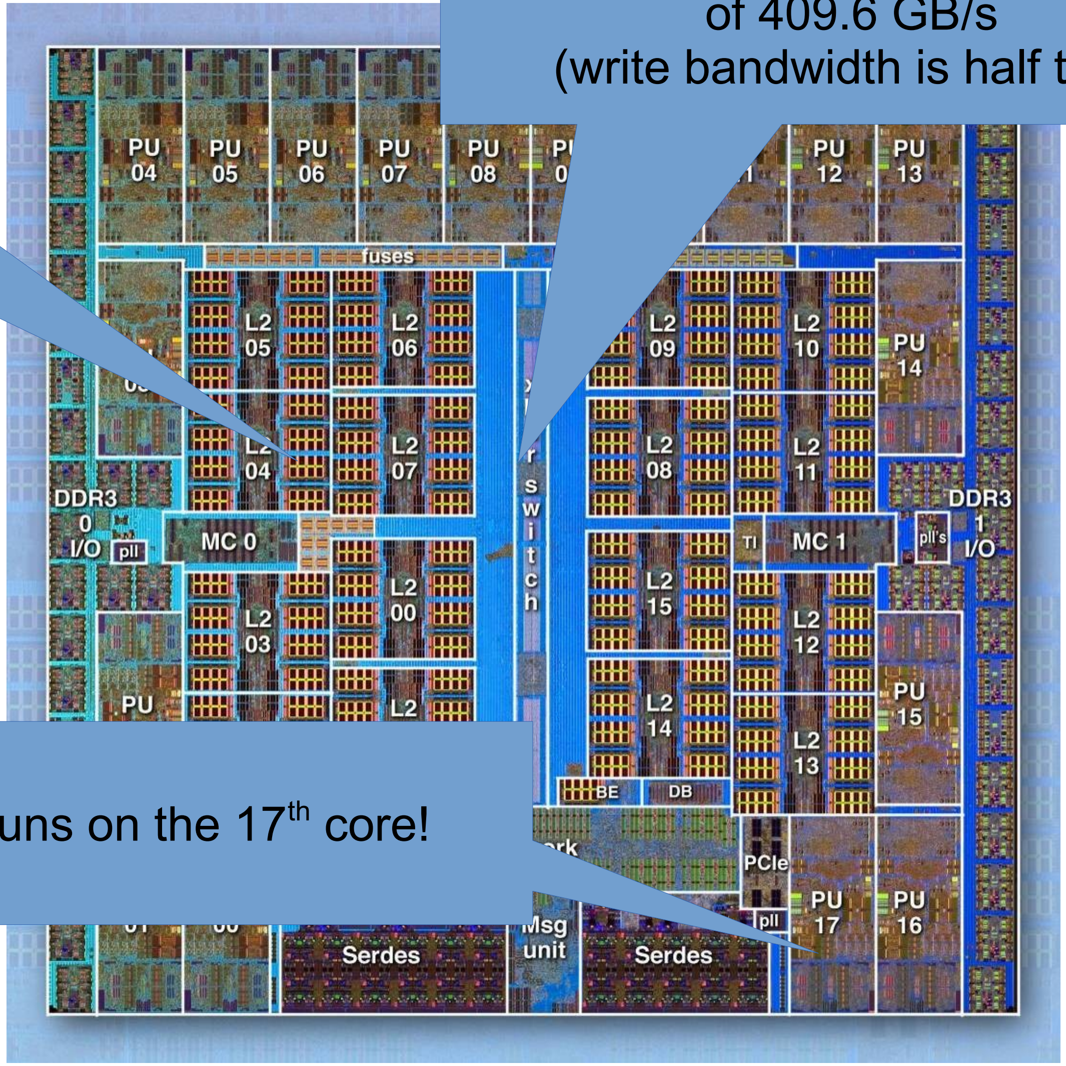


There are 16 cores per node

Each BG/Q CPU has 16 cores you can use

The cores are connected by a cross-bar interconnect with an aggregate read bandwidth of 409.6 GB/s (write bandwidth is half that)

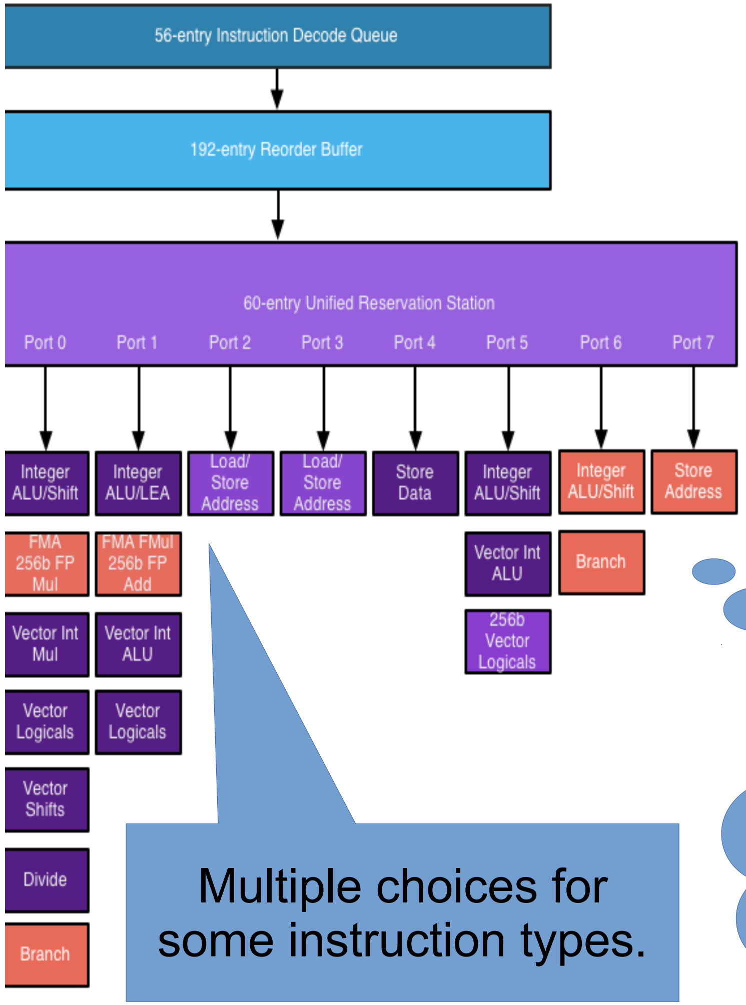


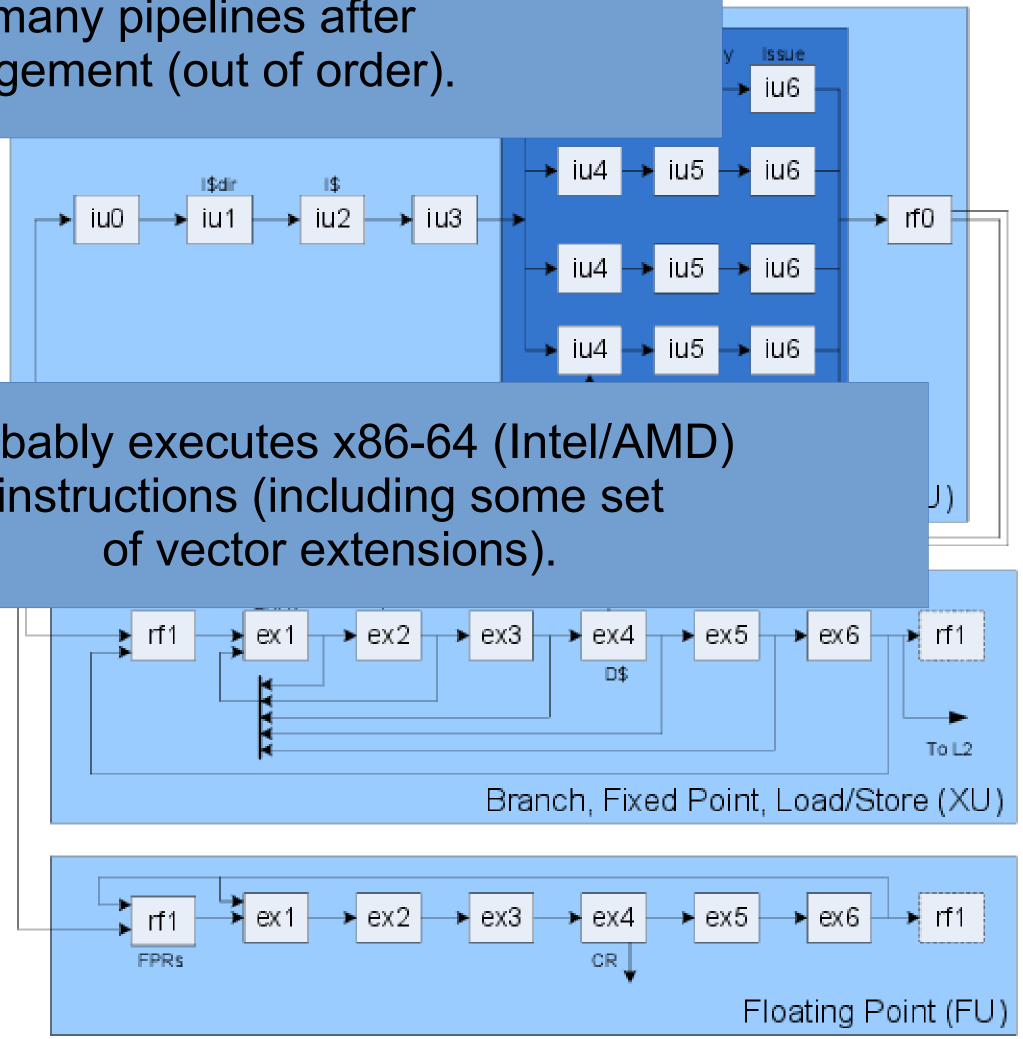


CNK, the lightweight operating system, runs on the 17th core!

Image source: https://computing.llnl.gov/tutorials/linux_clusters/

There are two pipelines per core





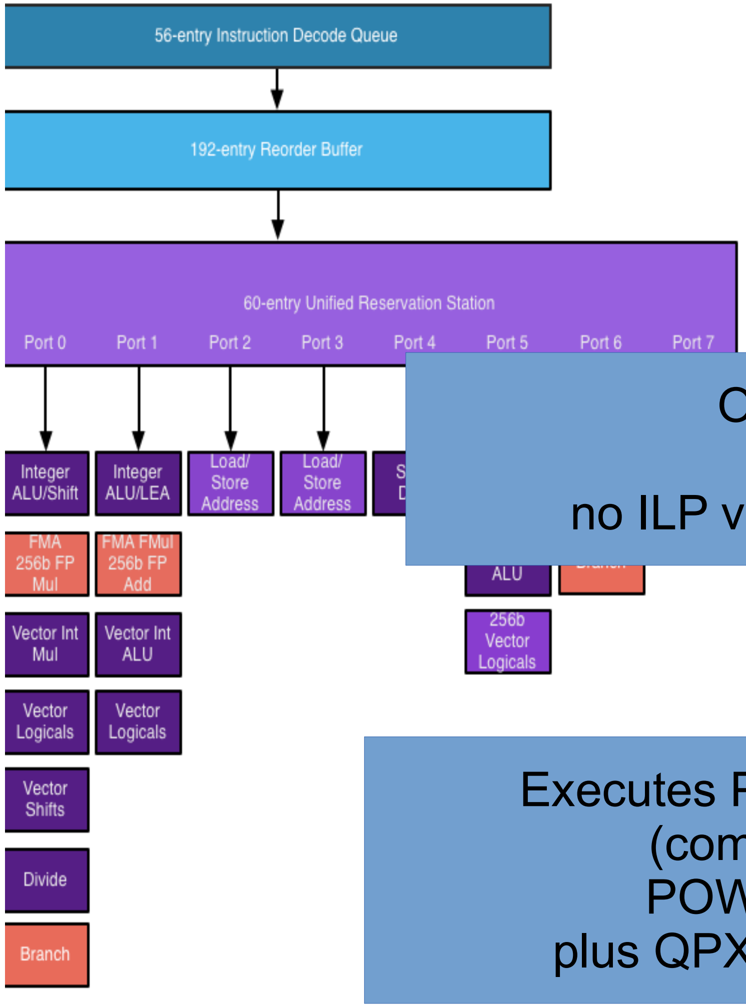
In commodity HPC cores, instructions are dispatched to many pipelines after dynamic rearrangement (out of order).

Probably executes x86-64 (Intel/AMD) instructions (including some set of vector extensions).

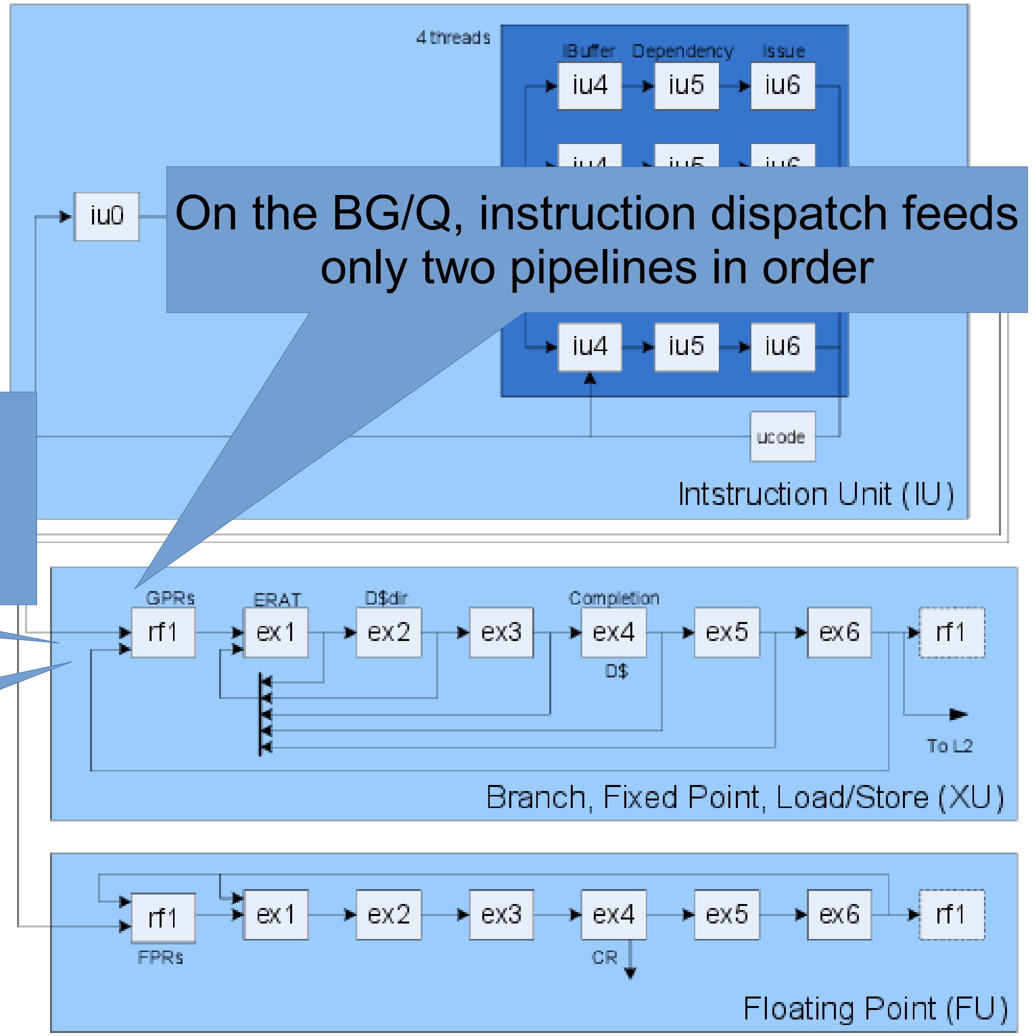
Multiple choices for some instruction types.

Not a BG/Q core

There are two pipelines per core



PowerPC A2 Core:



On the BG/Q, instruction dispatch feeds only two pipelines in order

Only one choice for any instruction: no ILP vs. vectorization tradeoffs!

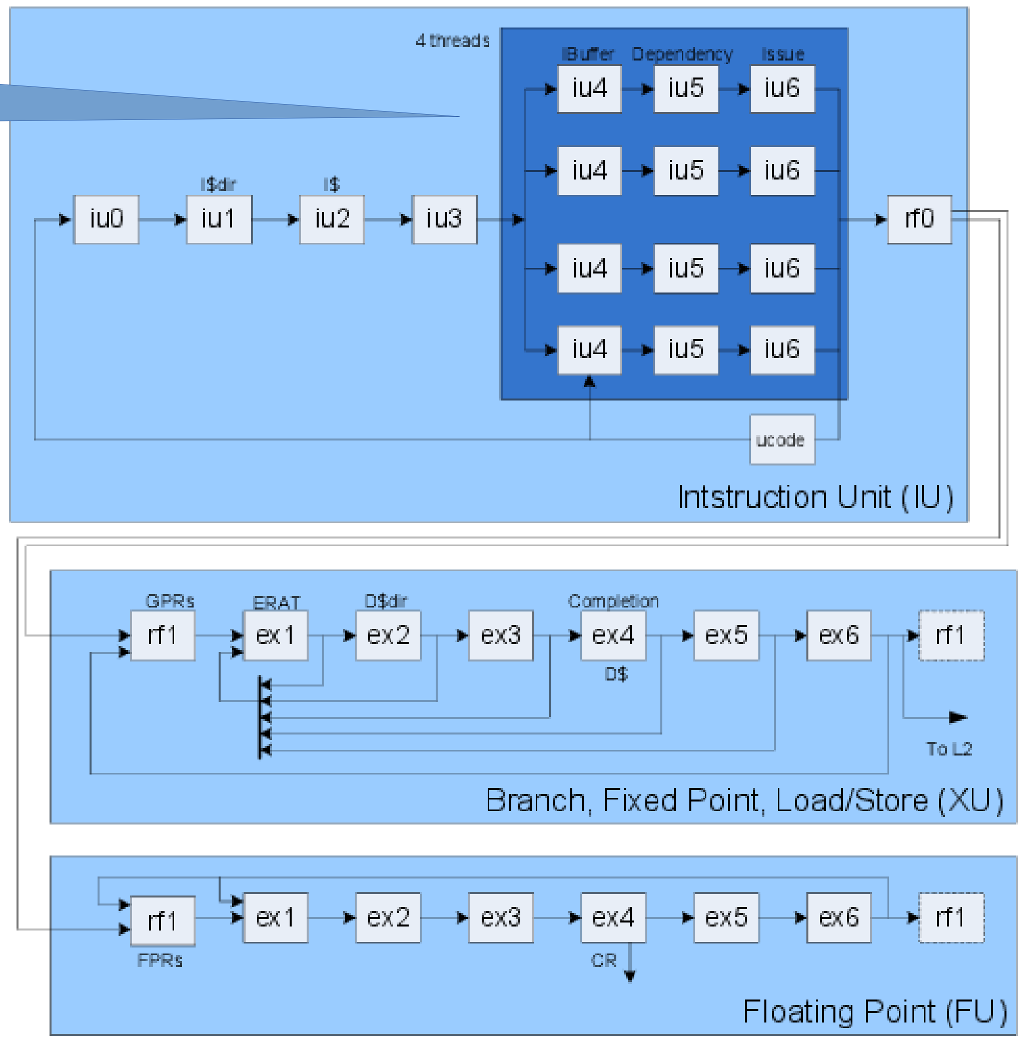
Executes PowerPC instructions (complying with the POWER ISA v2.06) plus QPX vector instructions

There are four hardware threads per core

Instructions from the four hardware threads are dispatched round-robin

The four threads share essentially all resources (except the register file)

The two pipelines can simultaneously start two instructions, but they must come from two different threads

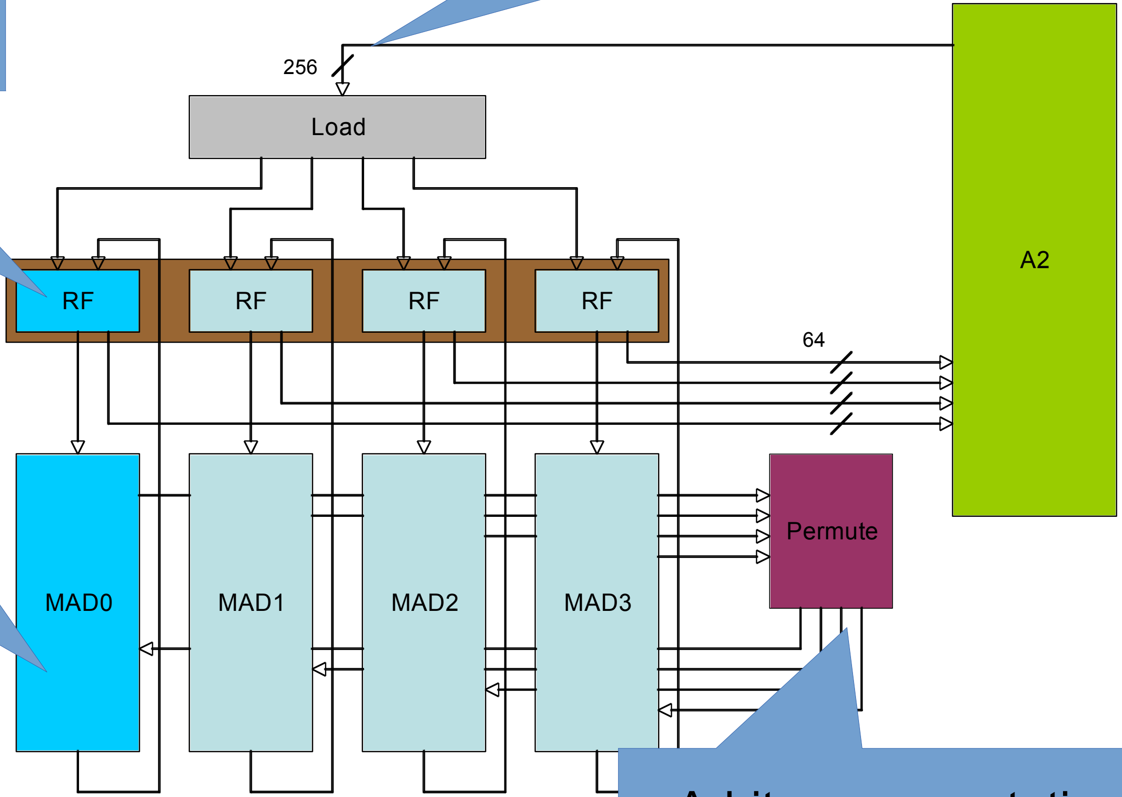


You must have at least two threads (or processes) per core to efficiently use the BG/Q!

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.



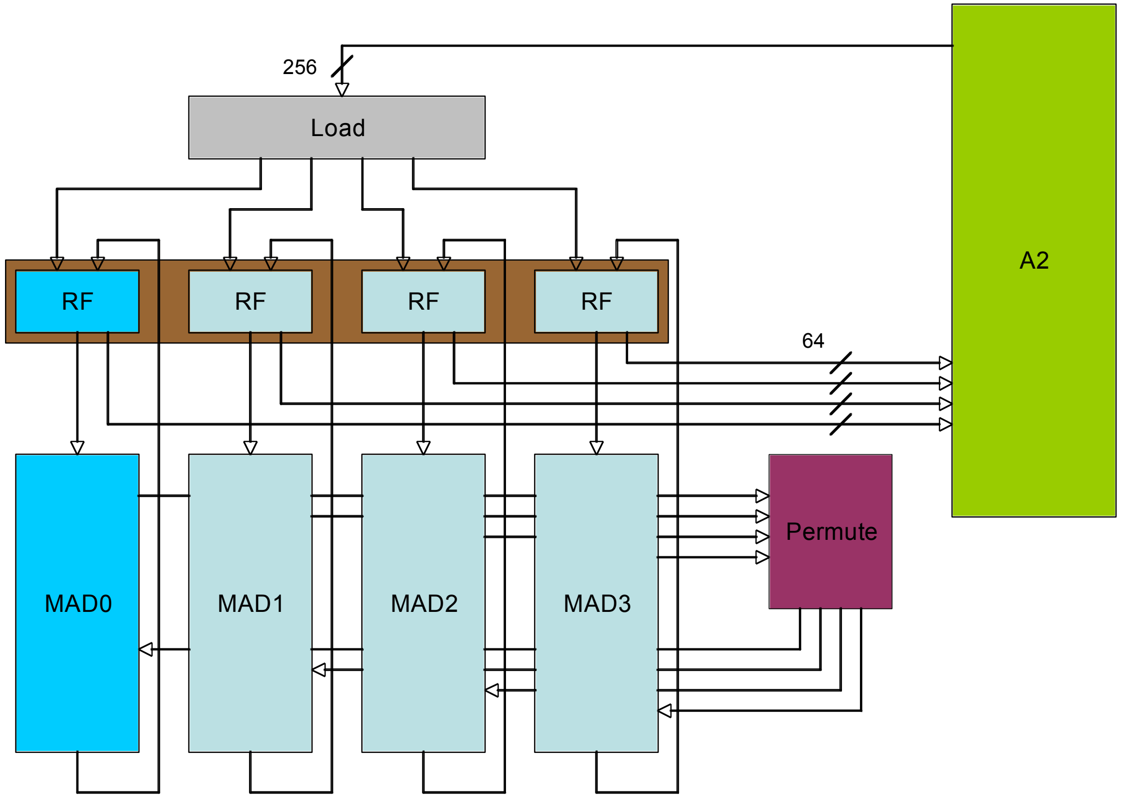
FP arithmetic completes in six cycles (and is fully pipelined). Loads/stores execute in the XU pipeline (same as all other load/stores).

Arbitrary permutations complete in only two cycles.

Vectorization: The Quad-Processing eXtension (QPX)

- On the BG/Q, only QPX vector instructions are supported!
- Only <4 x double>, <4 x float> and <4 x bool> operations are provided.
- The only advantage of single precision over double precision is decreased memory bandwidth/footprint.

On commodity HPC hardware, integer operations can also be vectorized, but not on the BG/Q.



Fused Multiply Add Instructions (FMA)

There are some FP (vector) instructions that combine both a multiply and an add/subtract into one instruction!

Many variants like these:

qvfmadd:
QRT0 ← [(QRA0)×(QRC0)] + (QRB0)
QRT1 ← [(QRA1)×(QRC1)] + (QRB1)
QRT2 ← [(QRA2)×(QRC2)] + (QRB2)
QRT3 ← [(QRA3)×(QRC3)] + (QRB3)

qvfmsub:
QRT0 ← [(QRA0)×(QRC0)] - (QRB0)
QRT1 ← [(QRA1)×(QRC1)] - (QRB1)
QRT2 ← [(QRA2)×(QRC2)] - (QRB2)
QRT3 ← [(QRA3)×(QRC3)] - (QRB3)

And a few like these with built-in permutations:

qvfxmadd:
QRT0 ← [(QRA0)×(QRC0)] + (QRB0)
QRT1 ← [(QRA0)×(QRC1)] + (QRB1)
QRT2 ← [(QRA2)×(QRC2)] + (QRB2)
QRT3 ← [(QRA2)×(QRC3)] + (QRB3)

qvfxxnpmadd:
QRT0 ← - ([(QRA1)×(QRC1)] - (QRB0))
QRT1 ← [(QRA0)×(QRC1)] + (QRB1)
QRT2 ← - ([(QRA3)×(QRC3)] - (QRB2))
QRT3 ← [(QRA2)×(QRC3)] + (QRB3)

Putting it all together...

You must vectorize to achieve
The peak FLOP rate
(on future machines, this factor
will be even larger)

You can only achieve the peak FLOP
rate using FMAs
(usually true on commodity hardware too)

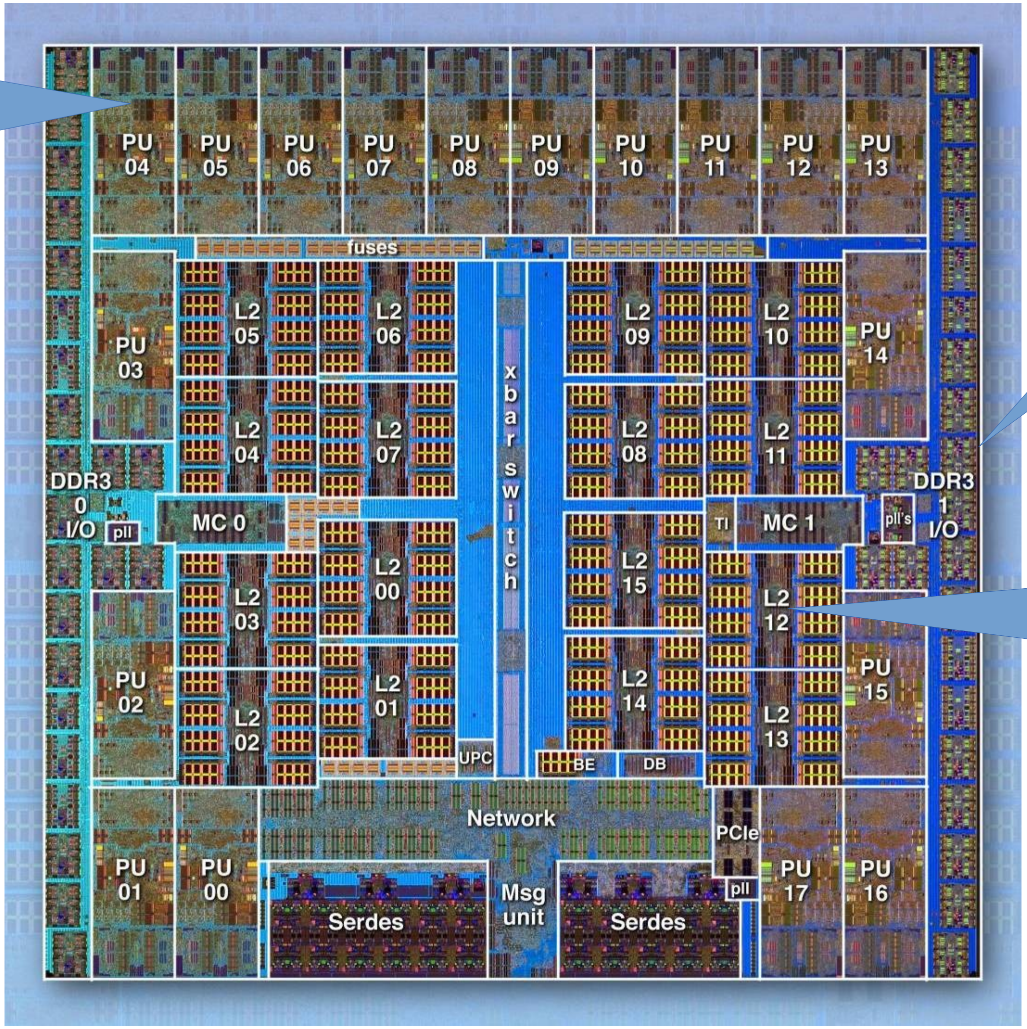
Peak FLOPS: (1.66 GHz) x (16 cores) x (4 vector lanes) x (2 operations per FMA) = 212.48 GFLOPS/node.

Remember you must use at least two
hardware threads (or processes)
or else you won't be able to
saturate the floating-point pipeline
in practice

Note: this is an order of magnitude
(on future machines, it will be nearly
two orders of magnitude)

Memory

L1 cache and L1P internal buffer (per core)

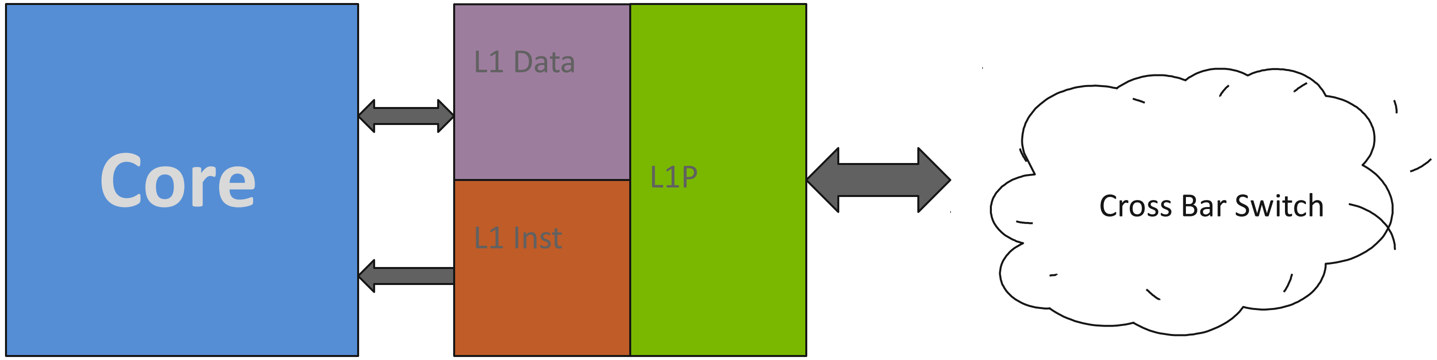


DDR3 DRAM (2 controllers)

Commodity HPC cores often also have an L3 cache; we don't. However, they have an L2 cache that is only hundreds of KB.

L2 cache (16 slices) 16 GB in total

The L1 cache and Prefetcher



- Each core has its own L1 cache and L1 Prefetcher (L1P)
- L1 Cache:
  - Data: 16 KB, 8-way set associative, 64-byte cache lines, 6-cycle latency
  - Instruction: 16 KB, 4-way set associative, 3-cycle latency
- L1 Pefetcher (L1P):
  - 32 buffer entries, 128 bytes each, 24 cycle latency
  - Buffer is write back
  - Operates in list or stream mode (stream mode is the default)
  - By default, tracks 10 streams x 3 128-byte cache lines deep

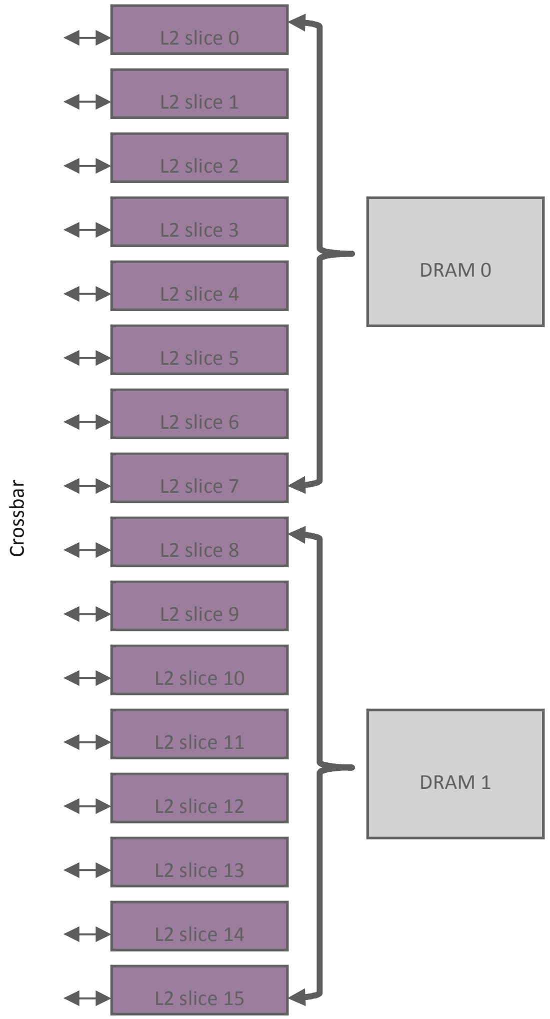
Hardware prefetching will never insert data directly into the L1 cache (data is stored in the L1P's buffer instead). Only explicit software prefetching can do that. The latency of reading from the L1P is still significant.

The L2 cache and DRAM

- L2 Cache:
  - Shared by all cores, divided into 16 slices
  - 32 MB total, 2 MB per slice
  - 16-way set associative, 128-byte lines, write-back, 82-cycle latency
  - Prefetches from DRAM based on L1P requests
  - Supports direct atomic operations
  - Supports multiversioning (for transactional memory)
  - Clocked at 800 MHz (half of the CPU rate)
  - Read: 32 bytes/cycle, Write: 16 bytes/cycle
- DRAM:
  - Two on-chip memory controllers, each connected to 8 L2 slices
  - Each controller drives a 16-byte DDR-3 channel at 1.33 Gb/s
  - The peak bandwidth is 42.67 GB/s (excluding ECC)
  - The latency is > 350 cycles

Twice the L1 line size

For high-performance locks and non-blocking data structures!



Odds and Ends

- ✓ The A2 core uses in-order dispatch, with one exception: There is an 8-entry load miss queue (LMQ) that holds loads and prefetches that miss the L1 cache, shared by all threads. Upon an L1 cache miss, the issuing thread does not actually stall until a use of the load is encountered.
- ✓ Try not to request the same L1 cache line more than once (especially relevant when using software prefetching); the second request will stall the thread until the first request is satisfied.
- ✓ The L2 cache is write-through (so writing to a cache line knocks it out of cache), so avoid writing to memory from which you soon expect to read. Unlike commodity hardware, which uses write-back caches, making write locality important, write locality is essentially irrelevant on the BG/Q.
- ✓ For a mispredicted branch, there is a minimum penalty of 13 cycles.
- ✓ If you need to compute 1/x (and don't need the exact IEEE answer) or 1/sqrt(x), QPX provides reciprocal estimate and reciprocal sqrt estimate functions. Combined with a Newton iteration or two, these give nearly-exact answers and are much less expensive than alternative methods.
- ✓ There is a timebase register on the A2 core which provides exact cycle counts. If you're trying to time something, use it!

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];
                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 want at least 2 threads per core.

Each statement requires 5 L1P streams, but we have only 10 per core shared among all threads.

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];
      }
      ...
}
```

The RHS expression is two dependent FMAs requiring at least 3 QPX registers (5 registers if we “preload” all of the input values). 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. This will require up to 3*5 == 15 QPX 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, but in case it doesn't...

An example...

```
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];
        }
        ...
```

These loads are not explicitly prefetched, so they'll be coming from the L1P buffer, not the L1 cache. We'll have ~24 cycles of latency, ~12 cycles/thread, to hide.

But, the compiler will probably “preload” the data for each iteration during the preceding iteration in order to hide this latency. If it does not, then you can perform this transformation manually, unroll more, etc.

Or you can insert software prefetch instructions using:
-qprefetch (a command-line flag)
__dcbt (an intrinsic function)
(these are for IBM's compiler; other compilers have different options)

An example...

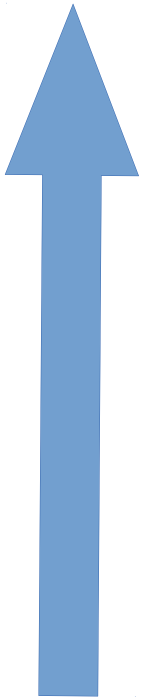
```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel
  {
#pragma omp for
#pragma unroll(3)
      for (i = 0; i < n; ++i) {
            a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
      }
      ...
```

But, the L2 can deliver only 32 bytes every two cycles,
so for the L2 to keep up, you want to do at least 2 QPX operations
for every loaded value. That would be 10 operations here, but we have only
2 FMAs + 5 loads + 1 store == 8 operations:
Only a higher-level change introducing more
data reuse can solve this problem!

Compiling

When compiling your programs, please use our MPI wrappers (these are the softenv keys)...

(generally best performance)



- ✓ +mpiwrapper-xl.legacy
- ✓ +mpiwrapper-xl
- ✓ +mpiwrapper-bgclang.legacy
- ✓ +mpiwrapper-bgclang
- ✓ +mpiwrapper-gcc.legacy
- ✓ +mpiwrapper-gcc

The “legacy” MPI gives the best performance unless you're using MPI_THREAD_MULTIPLE

bgclang has better C++ support than xl and gcc, but has no Fortran support (yet)

(generally worst performance)

Compiling

Basic optimization flags...

- ✓ -O3 – Generally aggressive optimizations (try this first: it is typically the best tested of all compiler optimization levels)
- ✓ -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)
- ✓ -qsmp=omp (xl) -fopenmp (bgclang and gcc) – Enable OpenMP (the pragmas will be ignored without this)
- ✓ -qnostrict (xl) -ffast-math (bgclang and gcc) – Enable “fast” math optimizations (most people don't need strict IEEE floating-point semantics). xl enables this by default at -O3 and above and you need to pass -qstrict to turn it off.

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

ESSL

IBM provides ESSL: A library of optimized math functions (BLAS for linear algebra, FFTs, and more). For FFTs, there is an optional FFTW-compatible interface.

- ESSL is installed in /soft/libraries/essl/current
- You can choose either -lesslbg or -lesslsmpbg (the 'smp' version uses OpenMP internally to take advantage of multiple threads)

ESSL is on IBM PowerPC systems what MKL is on Intel systems.

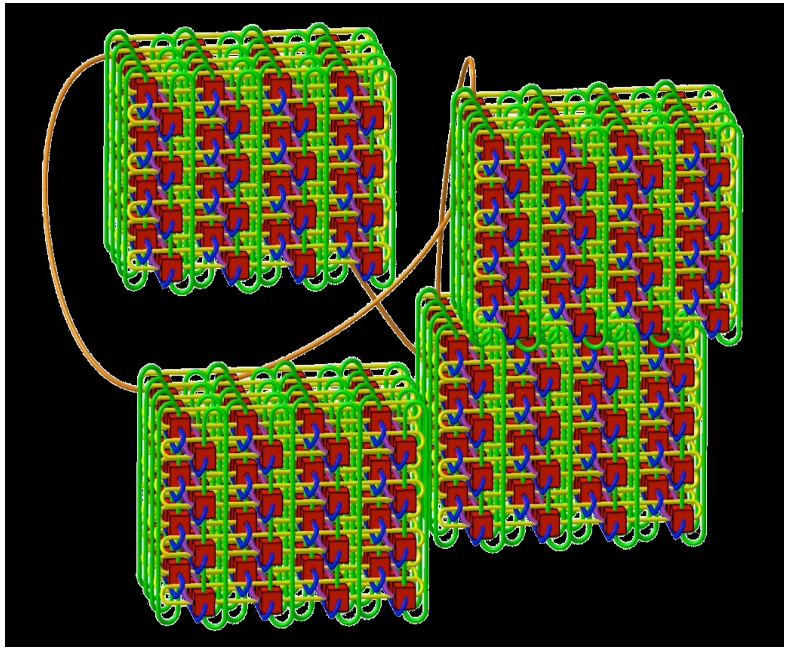
Memory partitioning

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

- Memory
  - Sending data between ranks on the same node often involves “unnecessary” copying
  - 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 16GB / (ranks per node) memory
- And parallelism
  - You'll likely see the best overall results from the scheme that exposes the most parallelism

Our 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

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

- ✓ Use MPI I/O (there'll be a talk on this later), 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 not at scale 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 BG/Q compilers... 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

Some final advice...

Don't guess! Profile! (We'll have several talks about how to do that.) Your performance bottlenecks on the BG/Q might be very different from those on other systems.

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

?

Any questions?