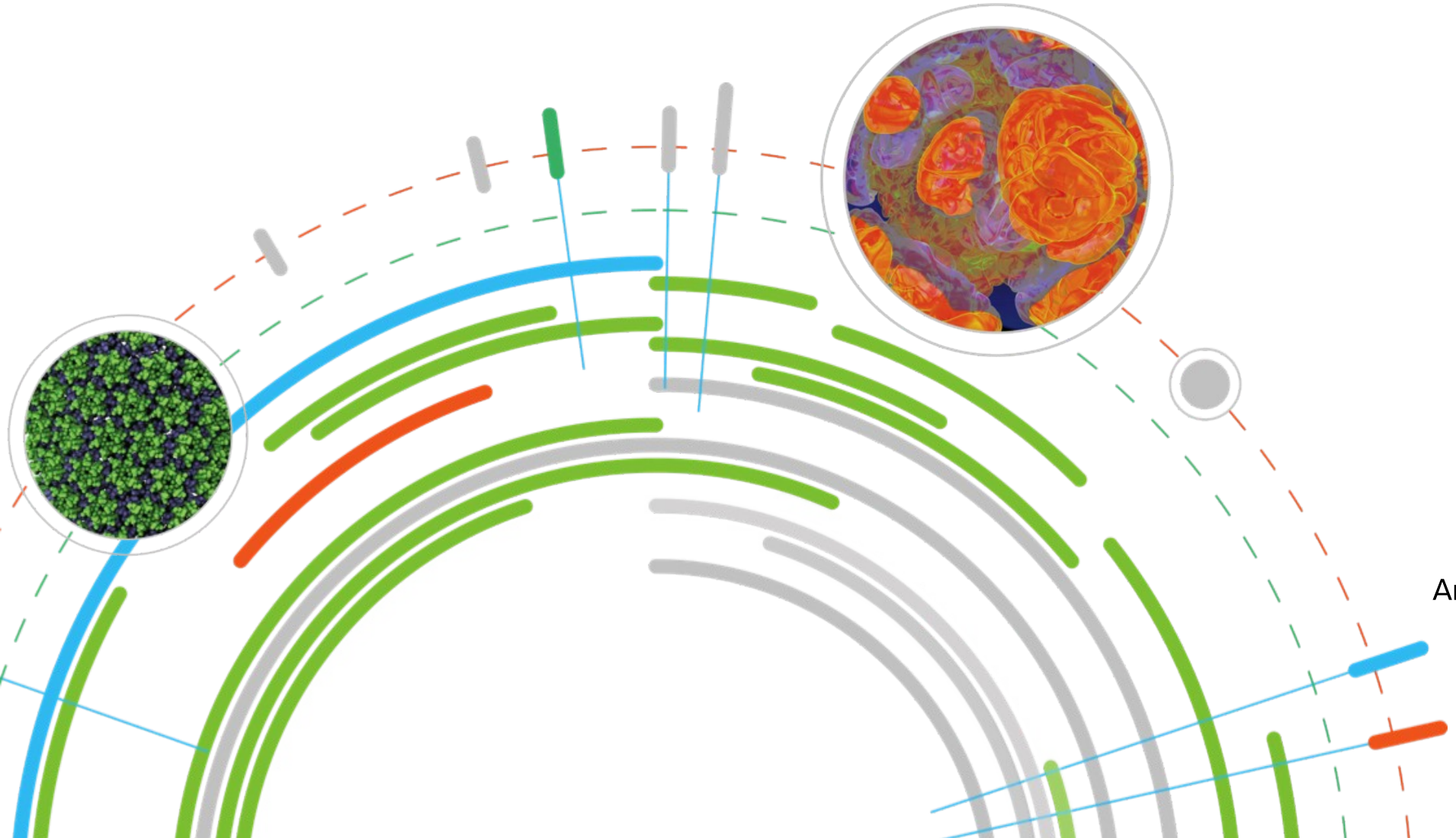
ALCF Systems 1: On-Node

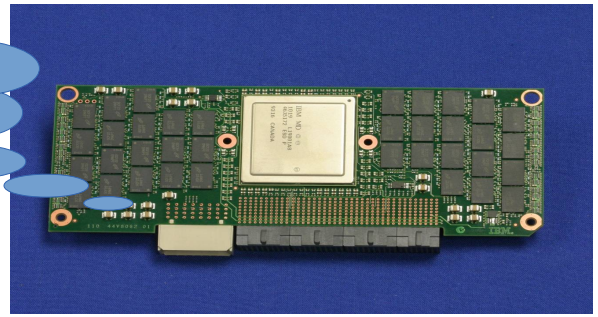


Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

Goals of This Presentation:

A BG/Q node looks like this:

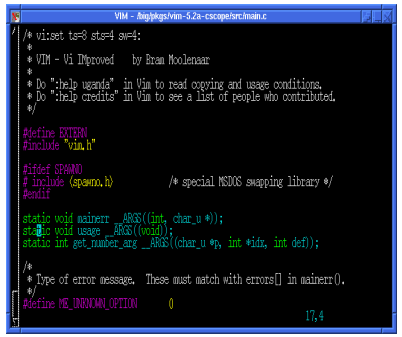


- ✔ To present the basic design and organization of the BG/Q node.

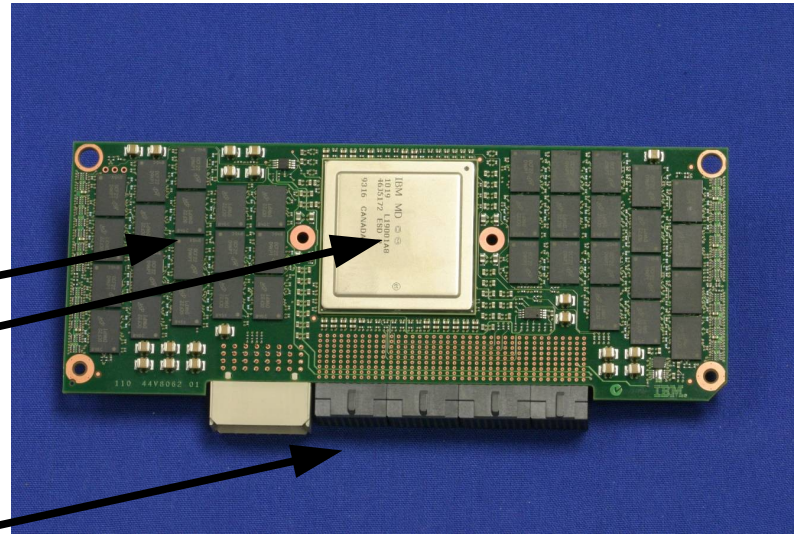
Not this:

- ✔ And compare with that found in commodity HPC hardware.

- ✔ Highlight aspects of the architecture relevant to code optimization.



Data Motion, Speed and Parallelism:



What does an application do:

- Retrieves data from memory
- Computes using that data
- Writes the results back into memory
- Interacts with other nodes, performs I/O, etc. (these are the subjects of later presentations)

The “speed” at which you can compute is determined by:

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

This is simple: 1.66 GHz

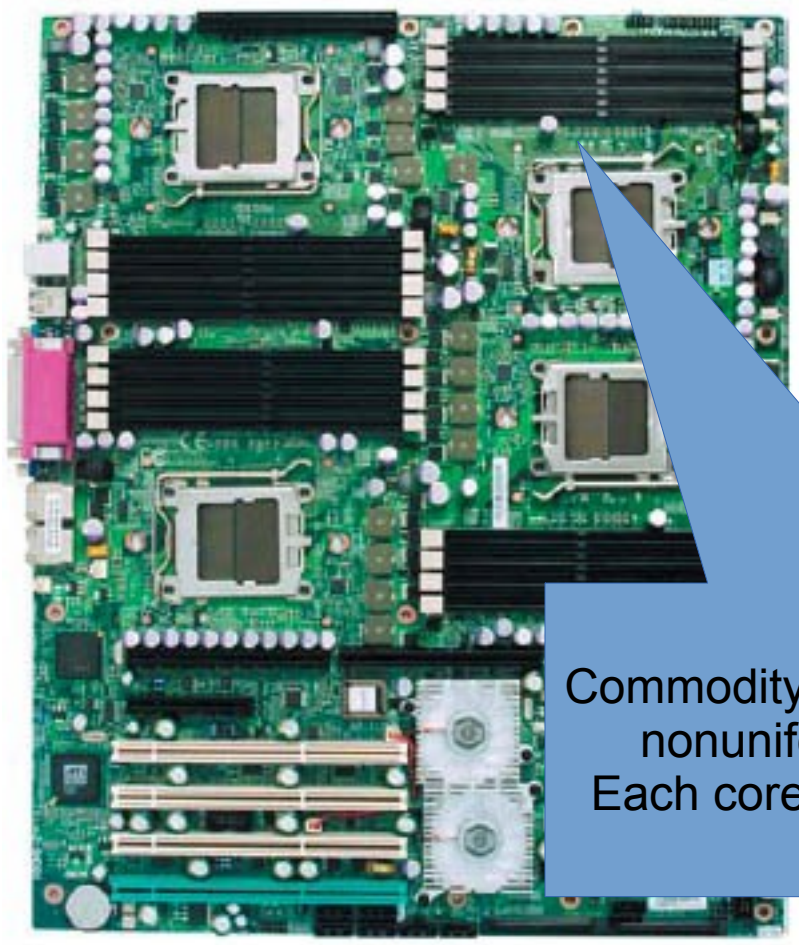
Types of Parallelism:

- Parallelism across nodes (using MPI, etc.) -- See upcoming presentations for information on this!
- 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
- Using instructions that perform multiple operations simultaneously (e.g. FMAs)

Hardware threads tie in here too.

This is similar to other systems for which you've optimized before, but there are important differences...

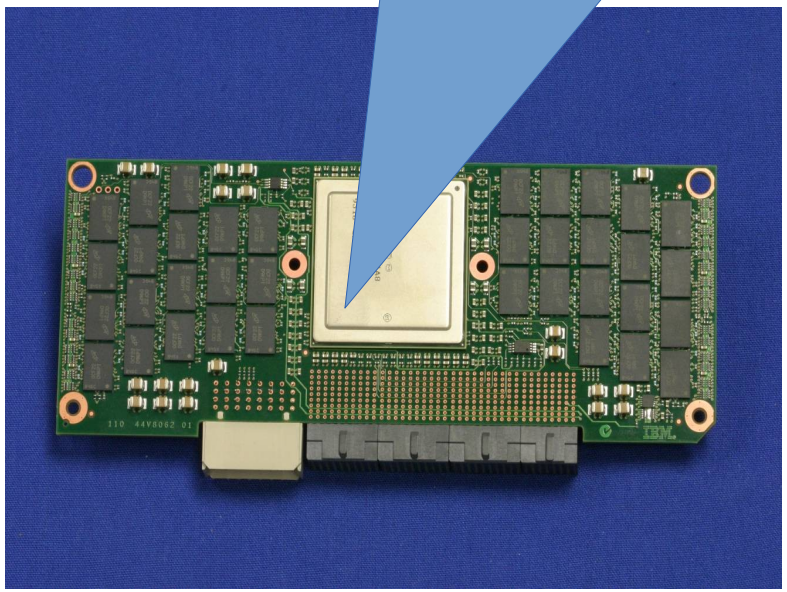
There Is Only One “Socket”:



Commodity HPC node with four sockets and nonuniform memory access (NUMA):
Each core has DRAM to which it is closer.

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

BG/Q: Only one “socket” with one CPU.
All memory is equally close:
No NUMA here!
(running only one MPI rank/node works well)



- Node has 1 CPU + 72 SDRAMs (16GB DDR3)
- Memory is soldered on for high reliability

There are 16 cores per node:

On the BG/Q, you'll want
(MPI process per node) x (threads per process)
to be at least 16 to run on all of the cores (as we'll discuss later, you'll probably want this number to be between 32 and 64).

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

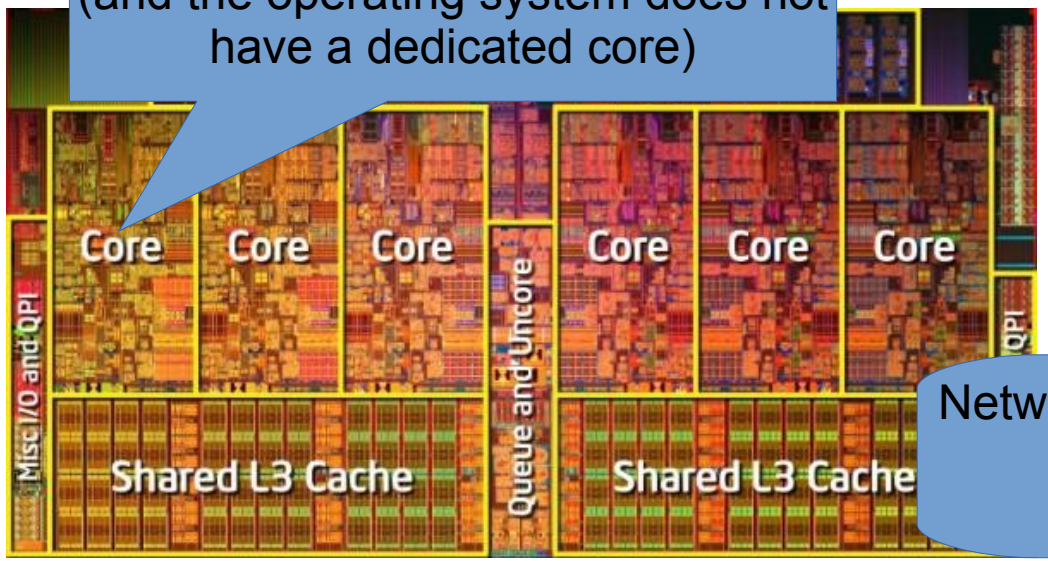
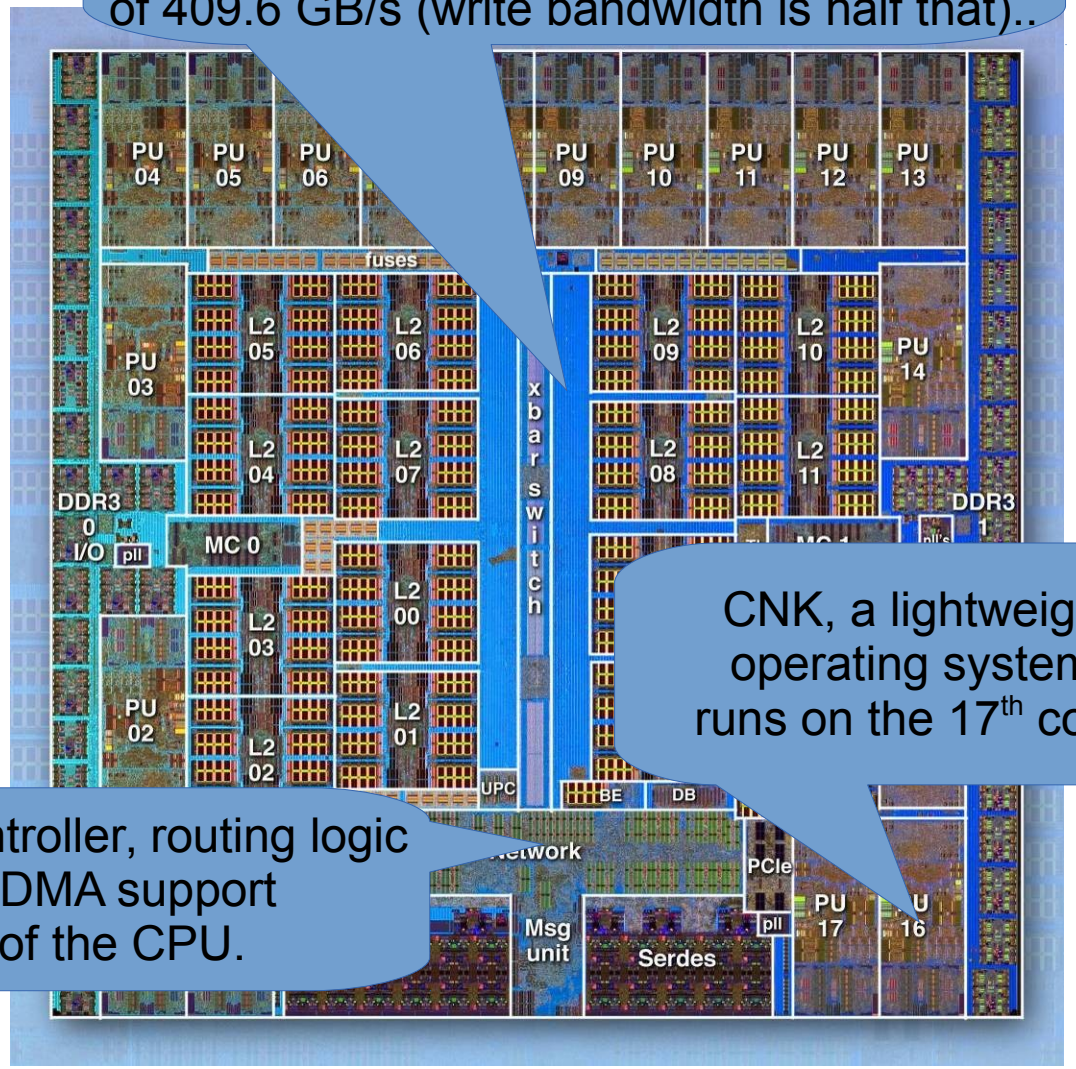


Image source: https://computing.llnl.gov/tutorials/linux_clusters/

Each BG/Q CPU has 16 cores connected by a cross-bar interconnect with an aggregate read bandwidth of 409.6 GB/s (write bandwidth is half that)..



CNK, a lightweight operating system, runs on the 17th core.

Network controller, routing logic and RDMA support part of the CPU.

There are two pipelines per core:

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

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

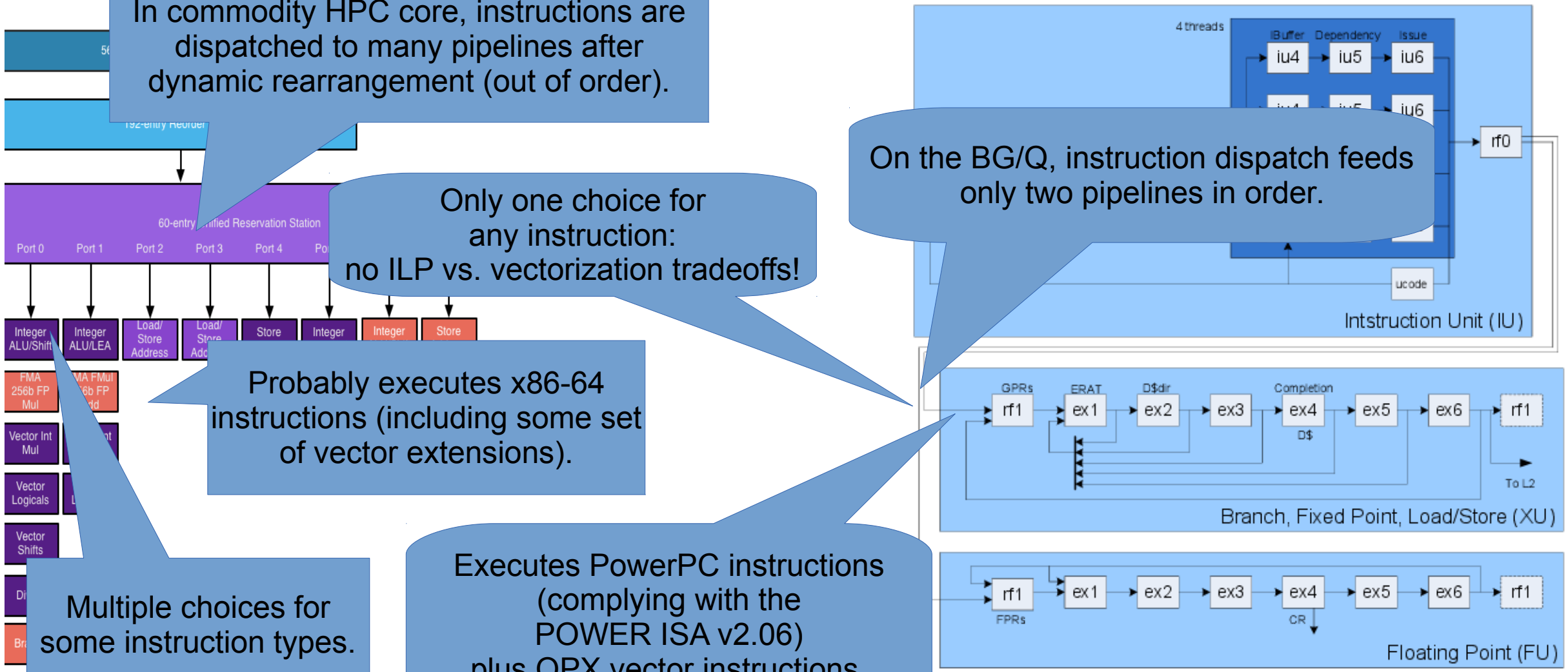
Probably executes x86-64 instructions (including some set of vector extensions).

Multiple choices for some instruction types.

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

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

Enhanced PowerPC A2 core:

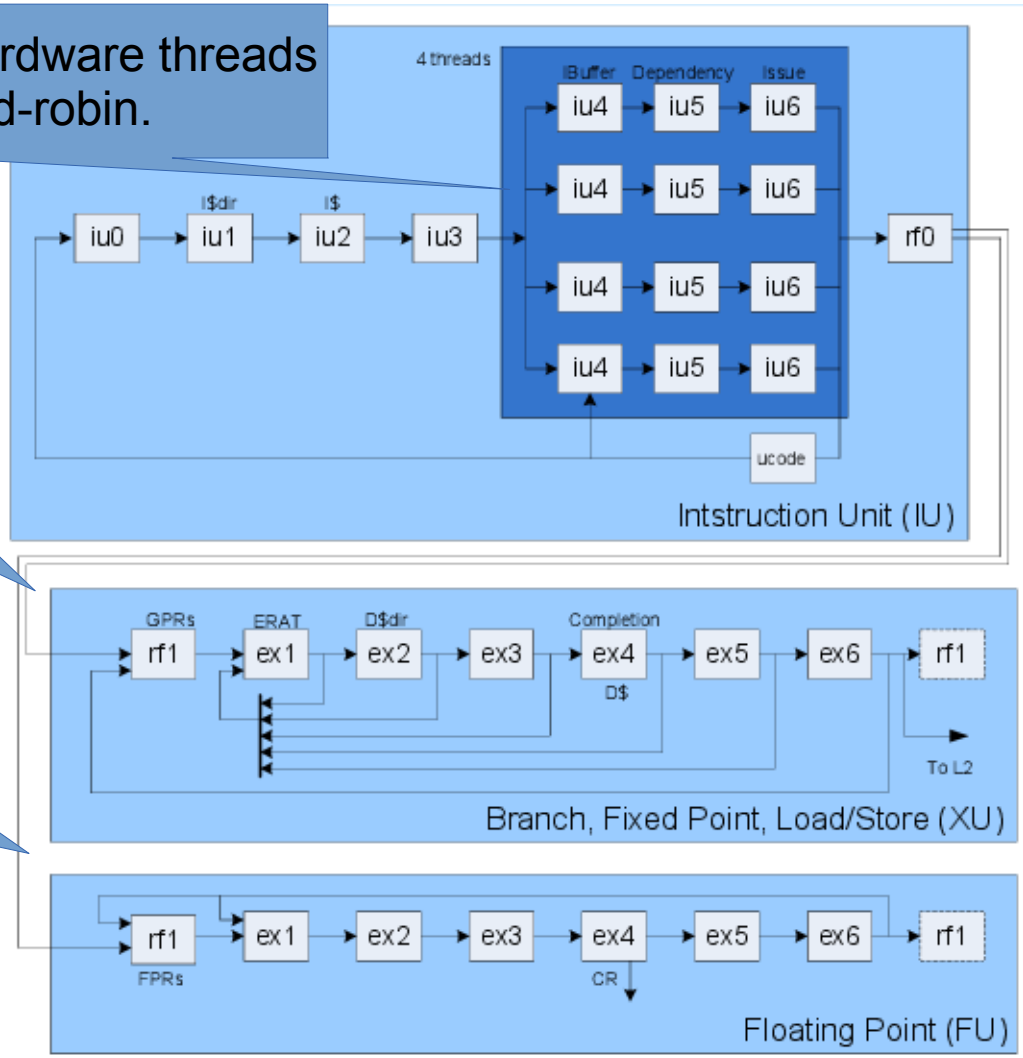


There are four hardware threads per core:

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.

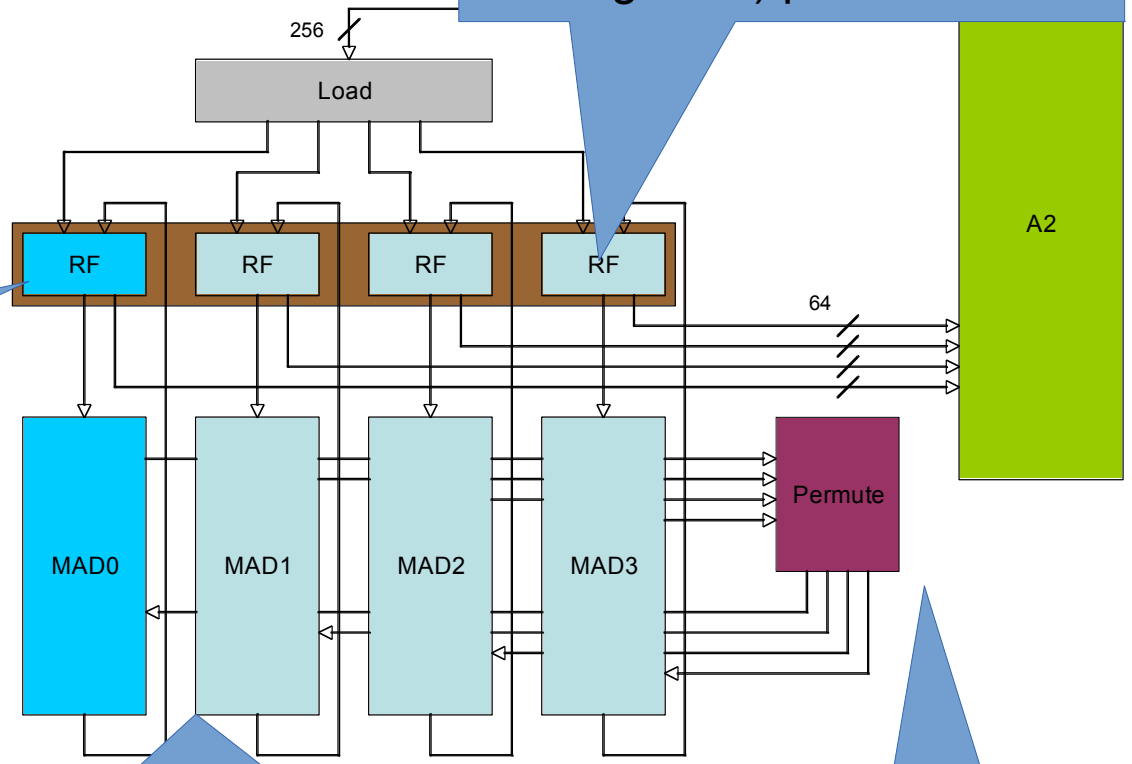


You must have at least two threads (or processes) per core to efficiently use the BG/Q!

Vectorization: Quad Processing eXtension (QPX):

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

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



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

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

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.

Fused multiply-add (FMA) instructions:

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

There are a few like this:

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 this:

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)

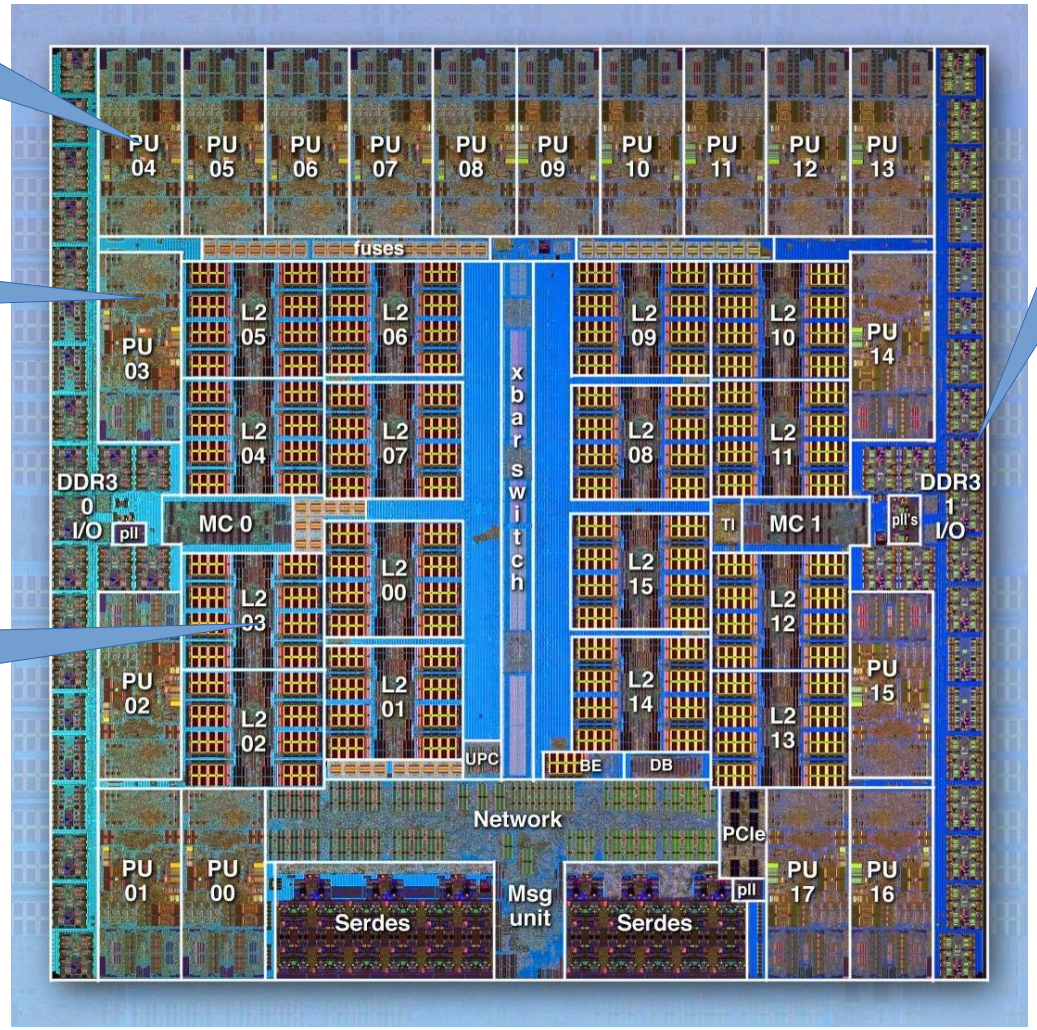
Peak FLOPS: (1.66 GHz) x (16 cores) x (4 vector lanes) x (2 operations per FMA) = 212.48 GFLOPS/node.

Memory Components:

L1 Cache (Per Core)

L1P Internal Buffer (Per Core)

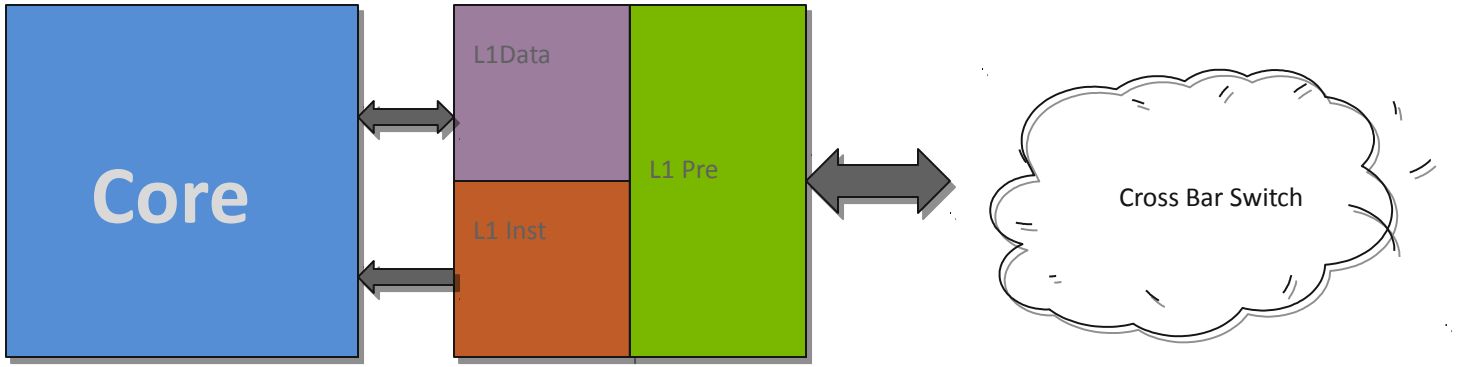
L2 Cache (16 slices)



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.

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 of stream modes (stream mode is the default)
  - By default, tracks 10 streams x 3 128-byte cache lines deep

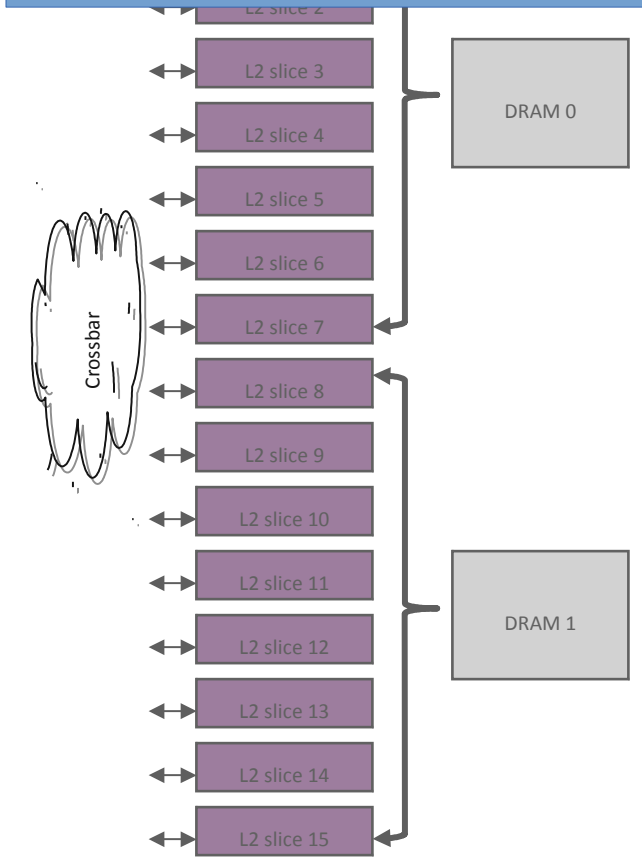
Hardware prefetching will never insert data directly into the L1 cache (data is stored in the L1P's buffer instead). Only explicit “software” prefetching can do that. The latency of reading from the L1P is still significant.

L2 Cache and Memory

- 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 @ 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 @ 1.33 Gb/s
  - The peak bandwidth 42.67 GB/s (excluding ECC)
  - The latency is > 350 cycles

This is twice the L1 cache line size.

If you have performance-critical locks, this is important!



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 a 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 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!

So what do you do with this information... An Example

We want at least 2 threads per core.

```
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.

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

Split loop body.

```
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];
      }
}
```

So what do you do with this information... An Example (cont.)

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

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 for
#pragma unroll(3)
      for (i = 0; i < n; ++i) {
            a[i]  = e[i]*(b[i]*c[i] + d[i]) + f[i];
      }
      ...
}
```

If the pragma does not do what you need, unrolling by hand is always an options.

So what do you do with this information... An Example (cont.)

```
void foo(double * restrict a, double * restrict b, etc.) {
#pragma omp parallel 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.

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!

.