

Performance Optimization I: Single Core/Node Vectorization, Memory - Overview and BG/Q

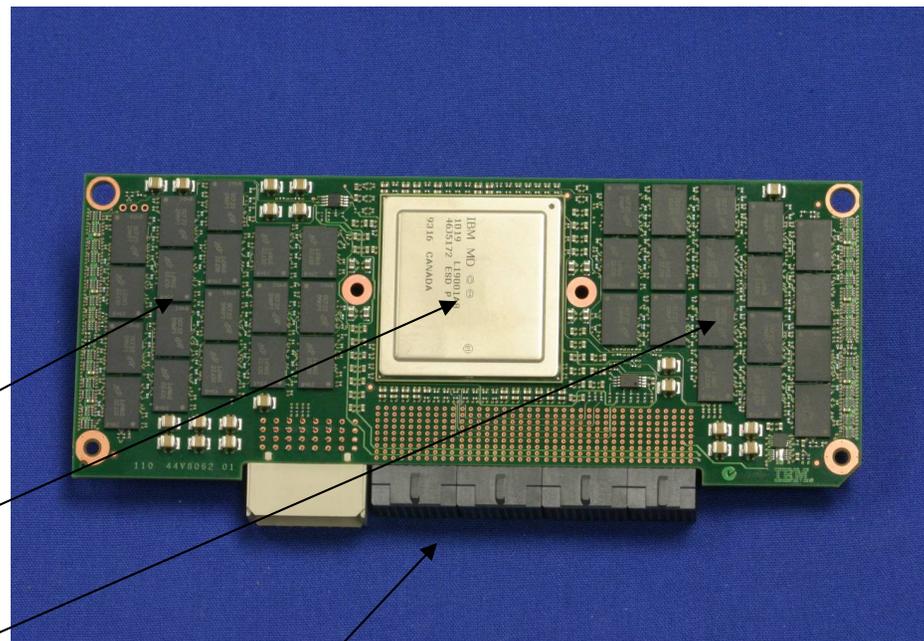


Scott Parker, Hal Finkel
Leadership Computing Facility

ALCF Computational Performance Workshop – May 15, 2018

What programs do...

```
IBM -> Application 5.2a: cccompilermain.c
/* visitor to @ start over:
 * VM - VS Improved by Ben Woodman
 * Do "help users" in VM to read copying and usage conditions.
 * Do "help credits" in VM to see a list of people who contributed.
 */
#define EXTEN
#include <math.h>
#define SPAN
#include <stdio.h> /* special MSIS swapping library */
#define
static void mainerr_RSS(int, char * *);
static void usage_RSS(int);
static int get_pointer_and_RSS(char * *p, int *idx, int def);
/* Type of error message. These suit match with errors[] in mainerr().
#define MS_UNREAL_OPTION 0 12,4
```

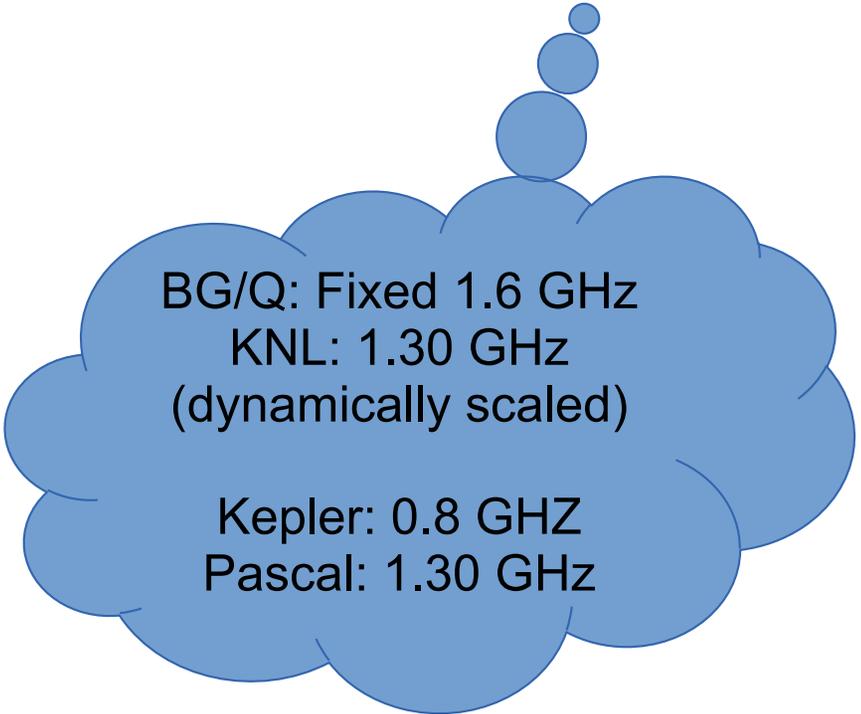


- ✓ Read data from memory
- ✓ Compute using that data
- ✓ Write results back to memory
- ✓ Communicate with other nodes and the outside world

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)



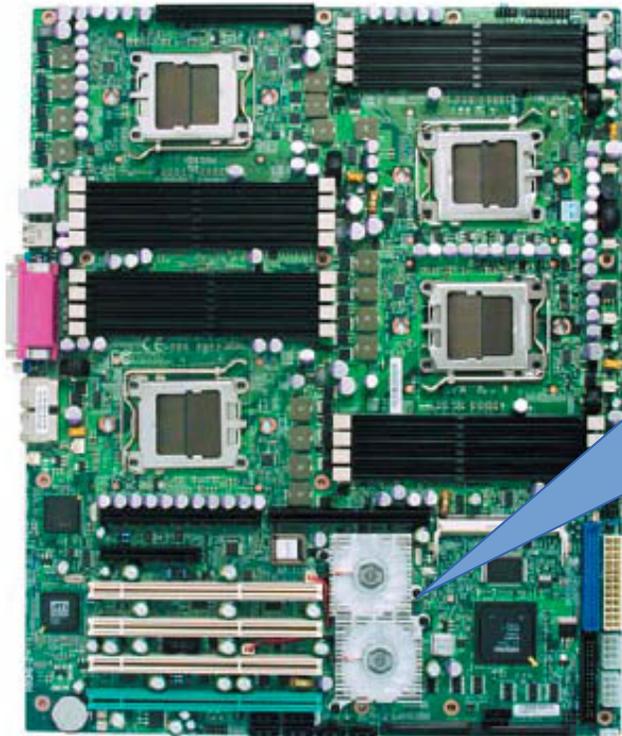
BG/Q: Fixed 1.6 GHz
KNL: 1.30 GHz
(dynamically scaled)

Kepler: 0.8 GHz
Pascal: 1.30 GHz



Your hard work goes here...

There is only one socket



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



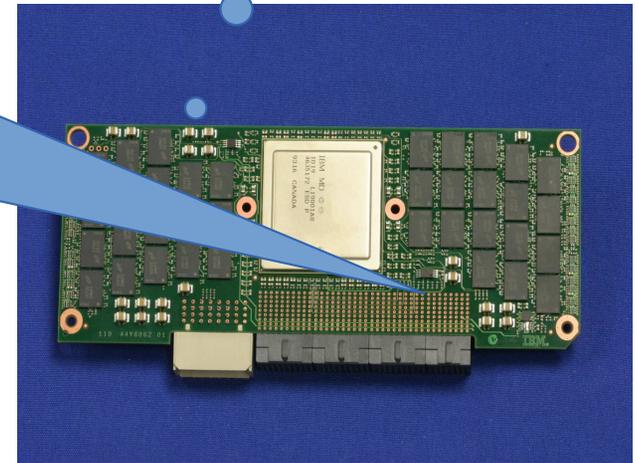
Image source: https://computing.llnl.gov/tutorials/linux_clusters/

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)

A BG/Q node



A BG/Q Node has:

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

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

There are 16 cores per node

Not a BG/Q core

Commodity HPC CPUs typically have only 4 – 24 cores (and the operating system does not have a dedicated core)

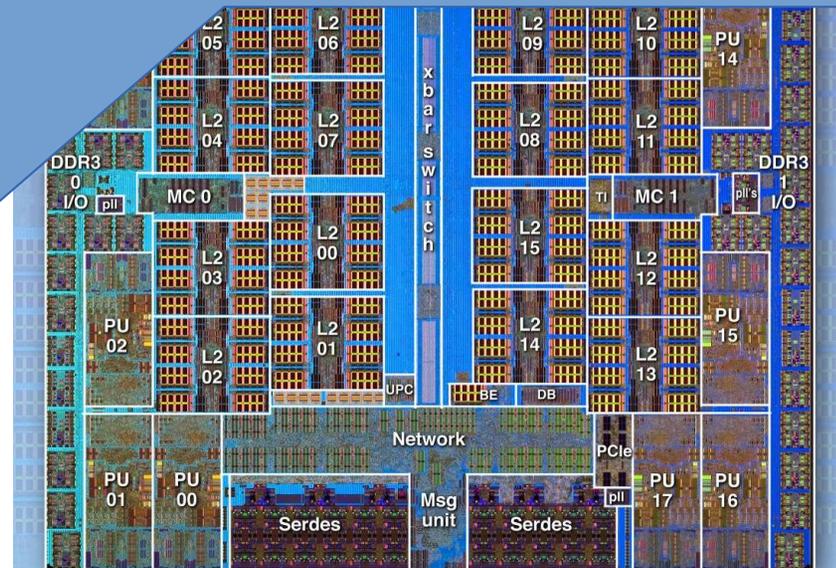
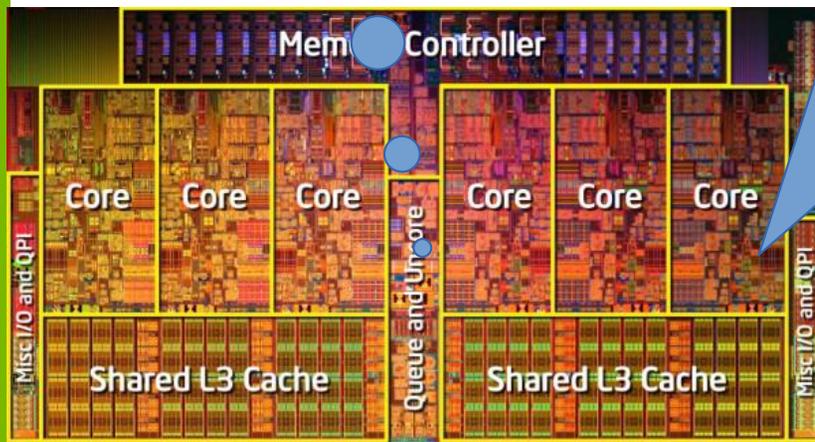
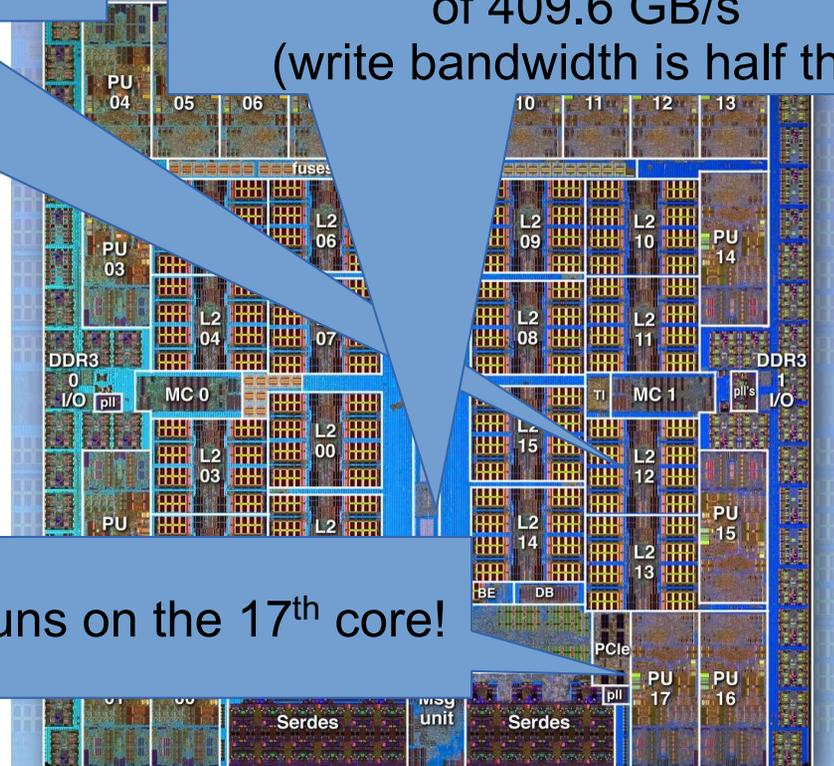
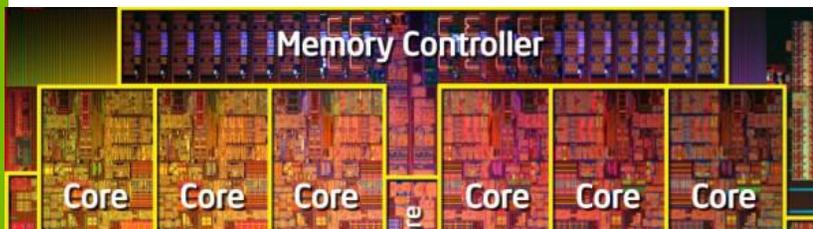


Image source: https://computing.llnl.gov/tutorials/linux_clusters/

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/

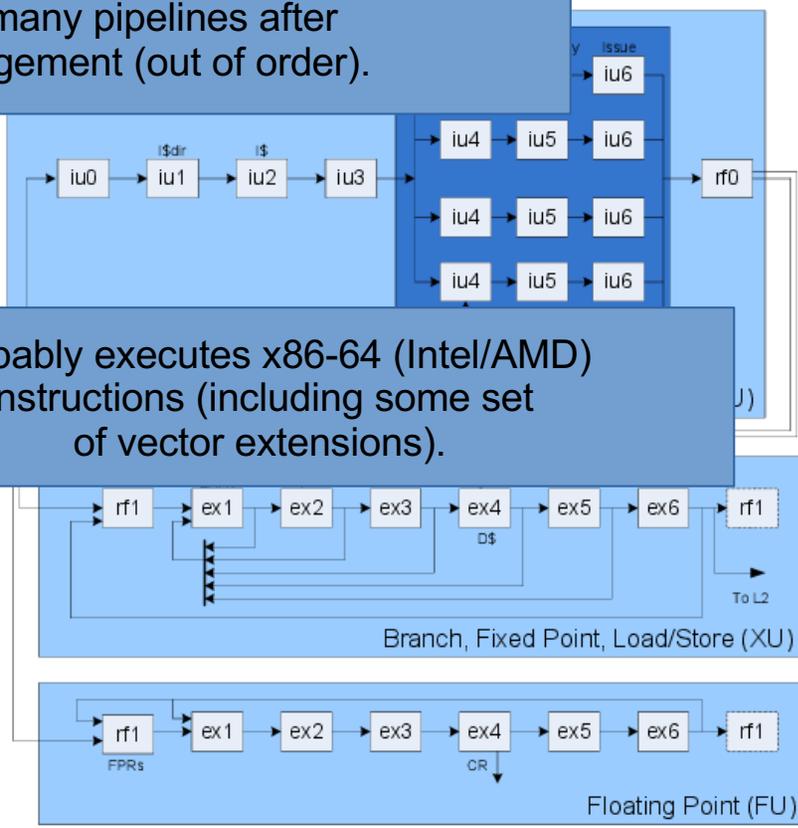
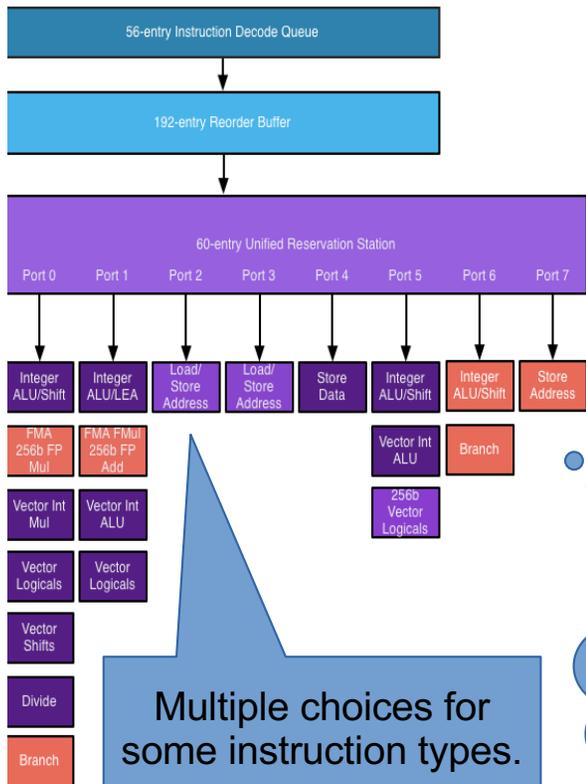
There are two pipelines per core

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

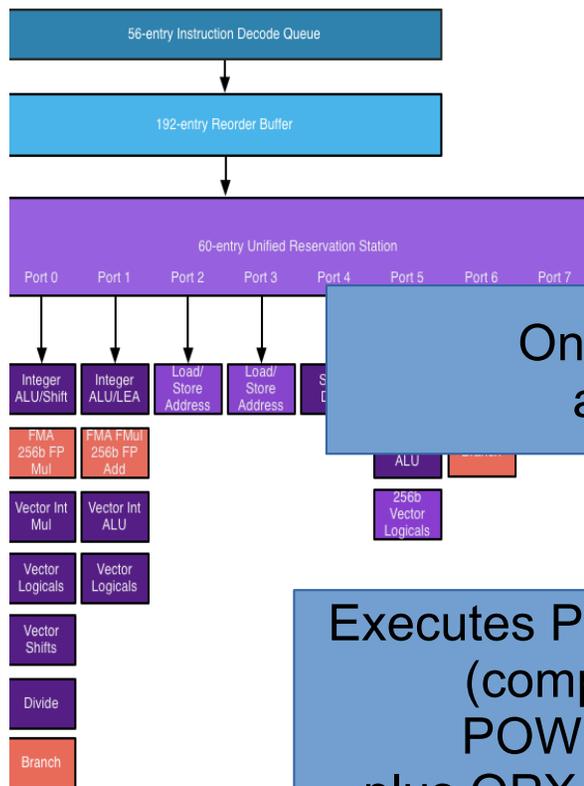
Probably executes x86-64 (Intel/AMD) instructions (including some set of vector extensions).

Not a BG/Q core

Multiple choices for some instruction types.



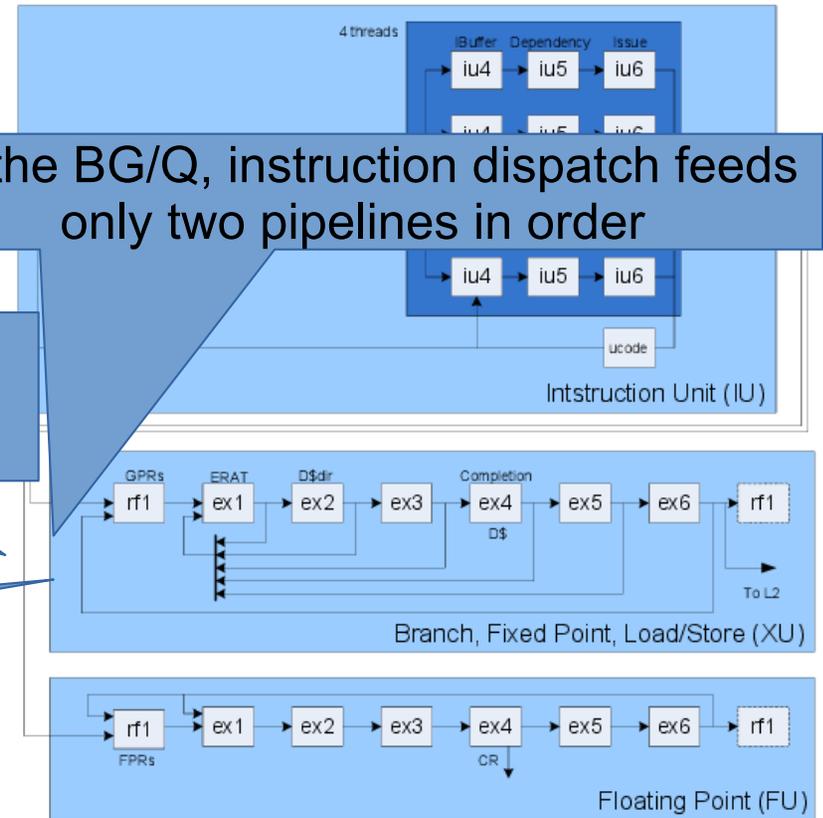
There are two pipelines per core



Only one choice for any instruction

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

PowerPC A2 Core:



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

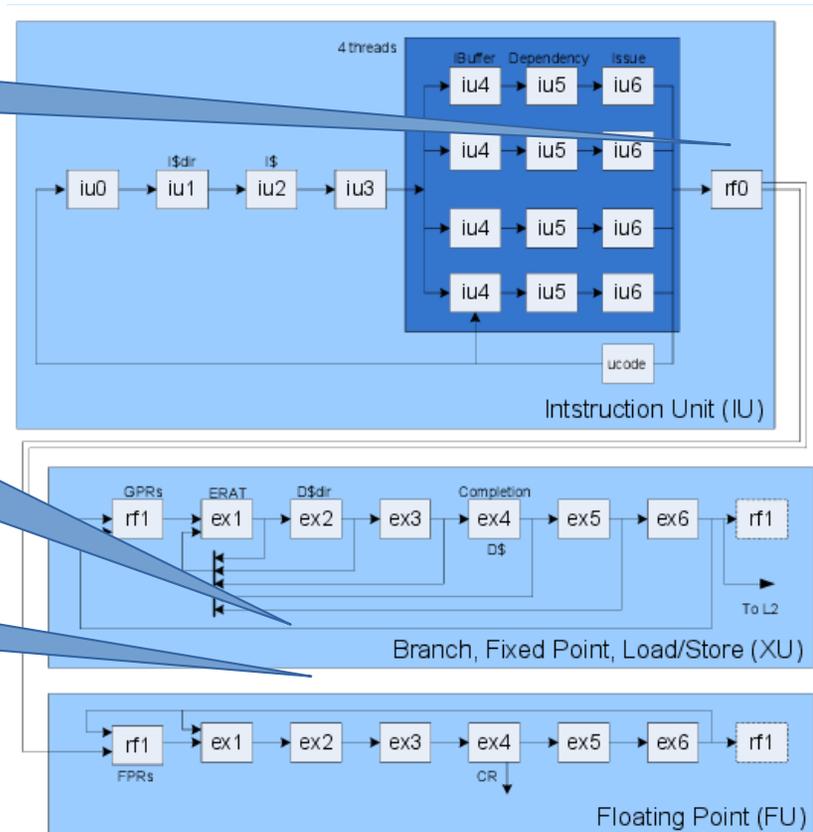
There are four hardware threads per core

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

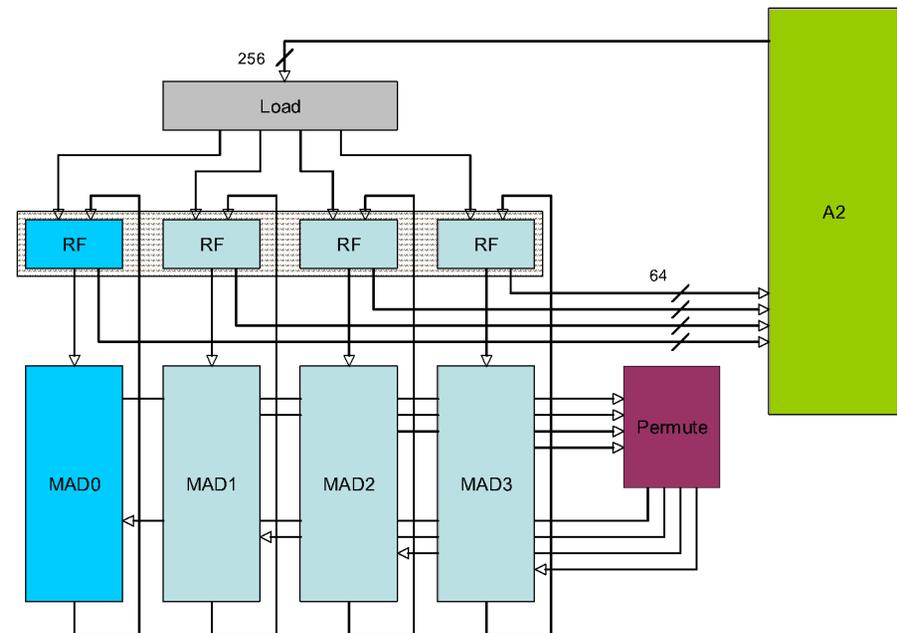
You must have at least two threads (or processes) per core to efficiently use the BG/Q!



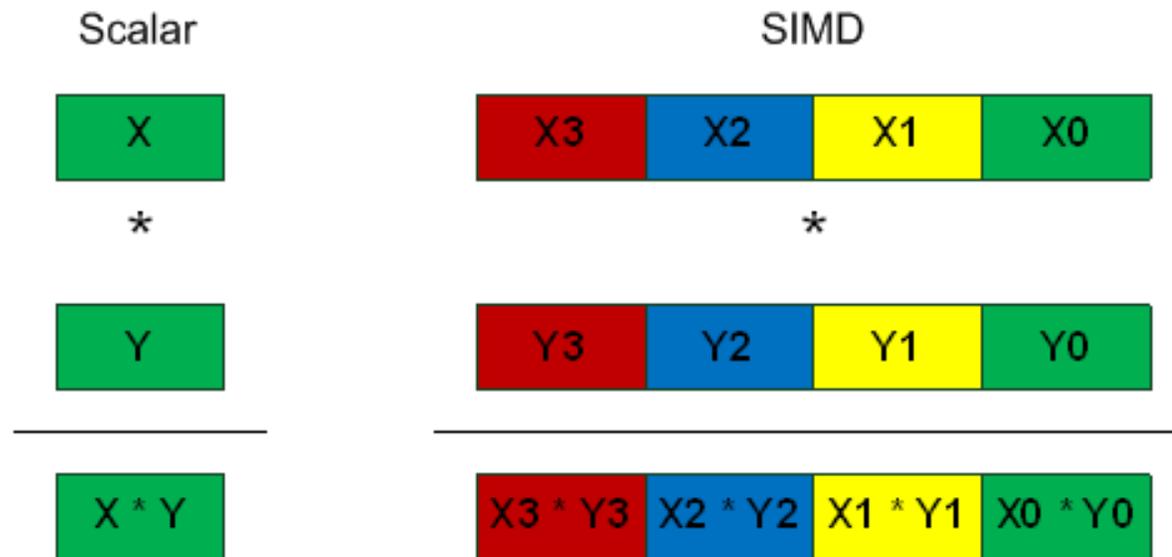
Vectorization: The Quad-Processing eXtension (QPX)

- ✓ 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.



SIMD: What does it mean?



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

Autovectorization (or manual vectorization)

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:

qvfmad:

$$\begin{aligned} \text{QRT0} &\leftarrow [(\text{QRA0}) \times (\text{QRC0})] + (\text{QRB0}) \\ \text{QRT1} &\leftarrow [(\text{QRA1}) \times (\text{QRC1})] + (\text{QRB1}) \\ \text{QRT2} &\leftarrow [(\text{QRA2}) \times (\text{QRC2})] + (\text{QRB2}) \\ \text{QRT3} &\leftarrow [(\text{QRA3}) \times (\text{QRC3})] + (\text{QRB3}) \end{aligned}$$

qvfmsub:

$$\begin{aligned} \text{QRT0} &\leftarrow [(\text{QRA0}) \times (\text{QRC0})] - (\text{QRB0}) \\ \text{QRT1} &\leftarrow [(\text{QRA1}) \times (\text{QRC1})] - (\text{QRB1}) \\ \text{QRT2} &\leftarrow [(\text{QRA2}) \times (\text{QRC2})] - (\text{QRB2}) \\ \text{QRT3} &\leftarrow [(\text{QRA3}) \times (\text{QRC3})] - (\text{QRB3}) \end{aligned}$$

And a few like these with built-in permutations:

qvfxmadd:

$$\begin{aligned} \text{QRT0} &\leftarrow [(\text{QRA0}) \times (\text{QRC0})] + (\text{QRB0}) \\ \text{QRT1} &\leftarrow [(\text{QRA0}) \times (\text{QRC1})] + (\text{QRB1}) \\ \text{QRT2} &\leftarrow [(\text{QRA2}) \times (\text{QRC2})] + (\text{QRB2}) \\ \text{QRT3} &\leftarrow [(\text{QRA2}) \times (\text{QRC3})] + (\text{QRB3}) \end{aligned}$$

qvfxxnfmadd:

$$\begin{aligned} \text{QRT0} &\leftarrow - ([(\text{QRA1}) \times (\text{QRC1})] - (\text{QRB0})) \\ \text{QRT1} &\leftarrow [(\text{QRA0}) \times (\text{QRC1})] + (\text{QRB1}) \\ \text{QRT2} &\leftarrow - ([(\text{QRA3}) \times (\text{QRC3})] - (\text{QRB2})) \\ \text{QRT3} &\leftarrow [(\text{QRA2}) \times (\text{QRC3})] + (\text{QRB3}) \end{aligned}$$

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: $(1.6 \text{ GHz}) \times (16 \text{ cores}) \times (4 \text{ vector lanes}) \times (2 \text{ operations per FMA}) = 204.8 \text{ GFLOPS/node}$.

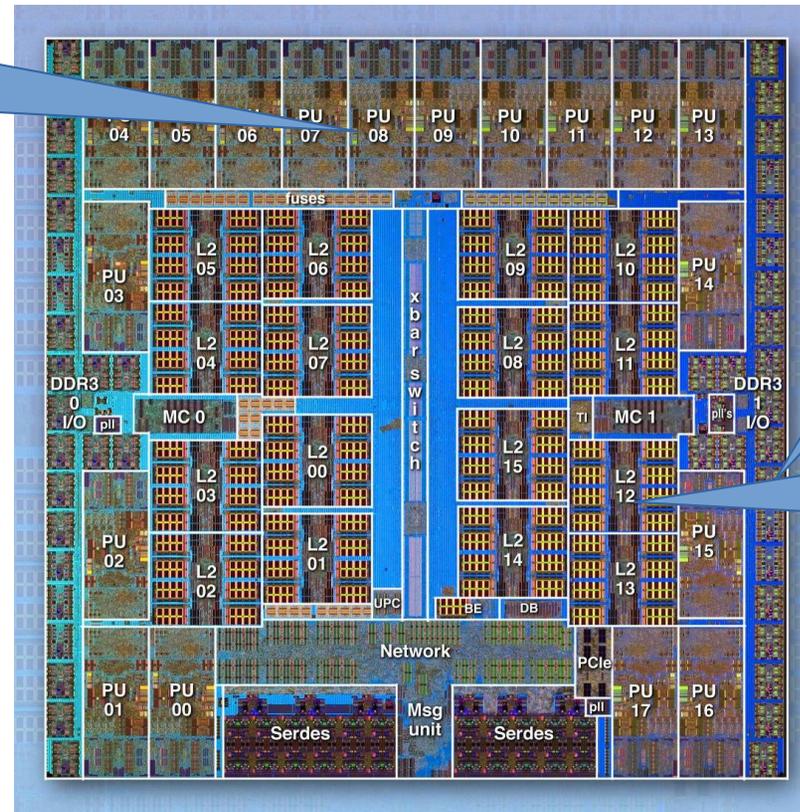
Remember you must use at least two
hardware threads (or processes)
or else you won't be able to
saturate the floating-point pipeline
in practice

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)

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



DDR3 DRAM
(~30 GB/s 2 controllers)

L2 cache
(16 slices)
16 MB in total

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!

Computer Architecture

Traditional computers are built to:

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

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$$

Computer Architecture

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

```
$ 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)

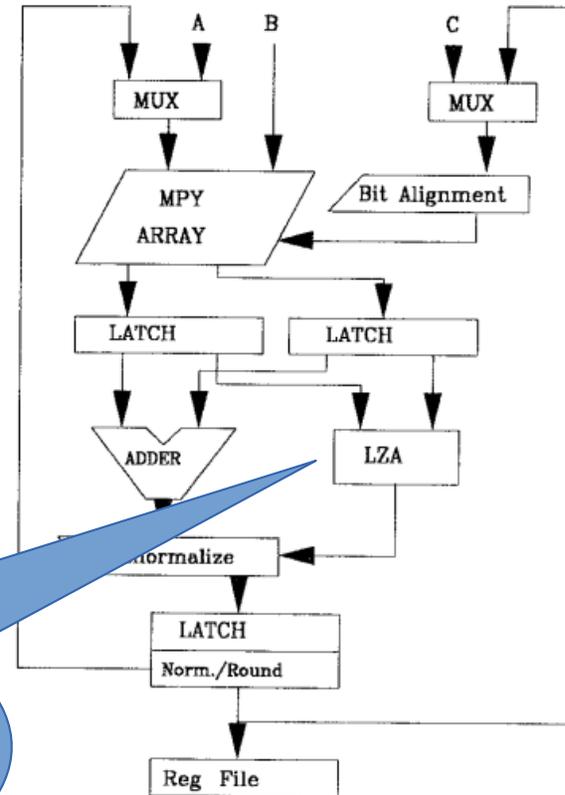
t1 = fma(t0, x, a2)

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.



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?

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

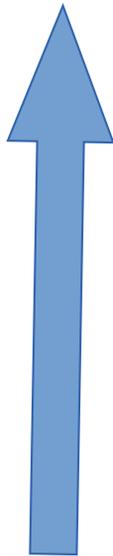
You can't unroll enough to completely hide anything but “on core” latencies (e.g. L1 cache hits and from FP pipeline) – you just don't have enough registers!

- 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)

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.

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.

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 (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

Some final advice...

Don't guess! Profile!

Your performance bottlenecks on the BG/Q might be very different from those on other systems.