The need for speed...

Bálint Joó,
Scientific Computing Group
Jefferson Lab

© KJ Pargeter * www.ClipartOf.com/1054524

Alternative Title:

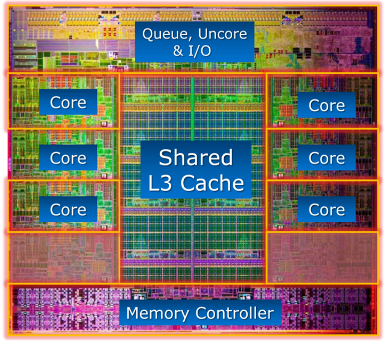
Reduce, Reuse, Recycle (as much as you possibly can)

Bálint Joó,
Scientific Computing Group
Jefferson Lab

Outline

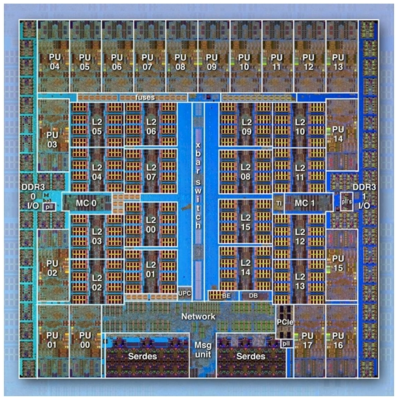
- Some features of modern computing systems
- Optimization and Constraints
- Performance: Limits and Modeling
- Multi-socket, multi-node issues
- Amdahl's Law and optimizations
- Summary

Parallel Processors



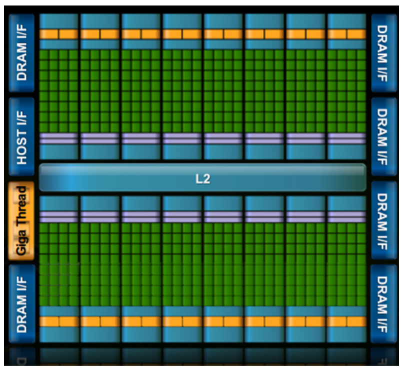
Intel Sandy Bridge E CPU
- 6 cores
- 256 bit AVX (8 single/4 double)
- 2 way Hyperthreading (SIMT)
- large shared cache
- 4 channel memory controller
- QPI to connect sockets/PCI etc

(image source: anandtech.com)



IBM BlueGene/Q
- 16 cores (+2 for system/etc)
- 256 bit vectors (4 double)
- 4 way SIMT
- L2 caches/cores connected with on chip crossbar network
- 2 DDR3 memory controllers

(image source: www.theregister.co.uk)

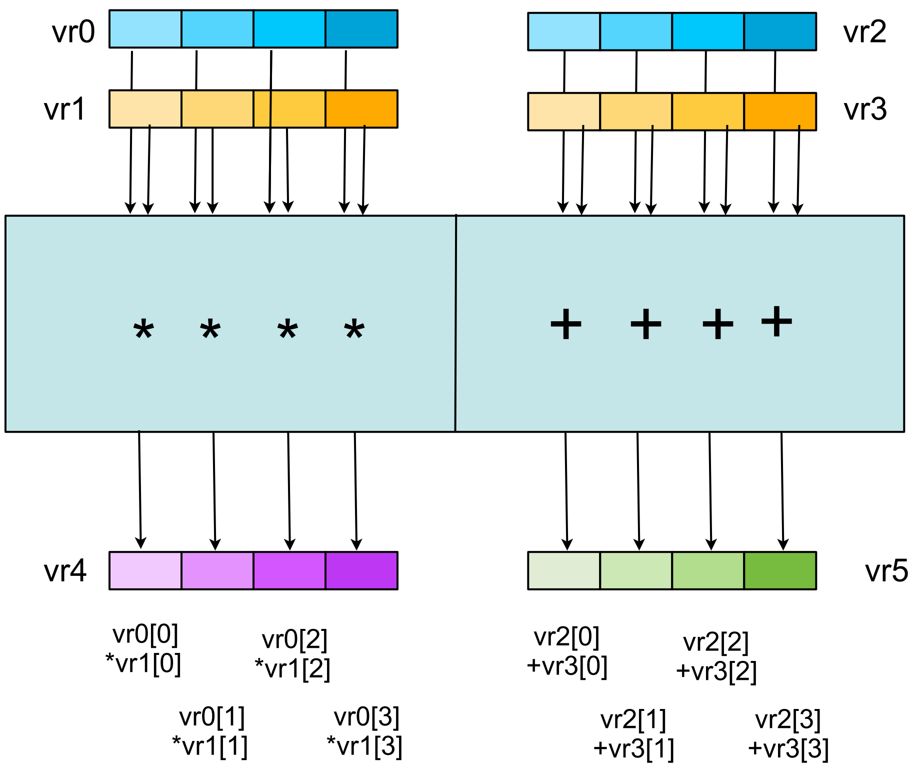


NVIDIA GF100 architecture
- 16 Symmetric Multiprocessor blocks
- 32 CUDA Cores per SM
- 2 sets of 16 way SIMT (per warp)
- Mike will have more details on this.

(image source: anandtech.com)

- Modern Computers
  - multiple sockets
  - SIMD vector units
  - multiple cores
  - co-processors
  - deep memory architectures
  - on and off chip communication networks
  - parallelism is ‘baked in’ (literally)

SIMD Vector Processing



- modern CPUs can usually work on more than one piece of data simultaneously
- Typically organize data as short 'vectors'
- Data is kept in 'vector registers' (vr here)
- Typically a CPU can perform a * and + simultaneously
- Some CPUs can also do FMA, ie: vr0*vr1+vr2

Do I need to learn assembler?

- Depends... I have not coded in assembler in a long time
- But I do find compiler intrinsics can be useful

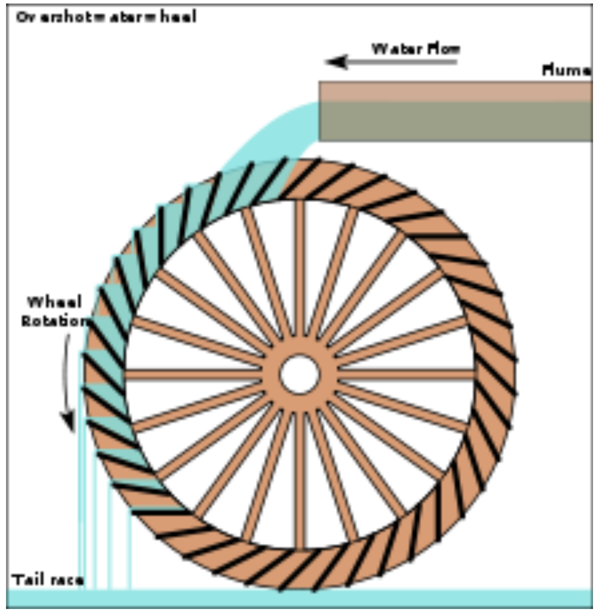
```
#include <xmmintrin.h> // SSE ops defined in this file

// a,b,c should be arrays of length 4, aligned on 16 byte
// boundaries. Routine does  a*b + c
void fmadd4( float* a, float* b, float* c )
{
   __m128 av, bv, cv;    // SSE registers (vector length=4)
   av = _mm_load_ps(a); bv=_mm_load_ps(b); cv=_mm_load_ps(c);
   cv = _mm_add_ps(cv,_mm_mul_ps(av,bv));
   _mm_stream_ps(c, cv);
}
```

  - Some compilers are better at vectorization than others.
    - may do as good a job as you writing intrinsics in some instances.
  - Intrinsics can tie you to specific vector length
    - above is SSE (length 4). Modern x86 machines can do AVX (length 8)
  - Intrinsics may tie you to specific compilers

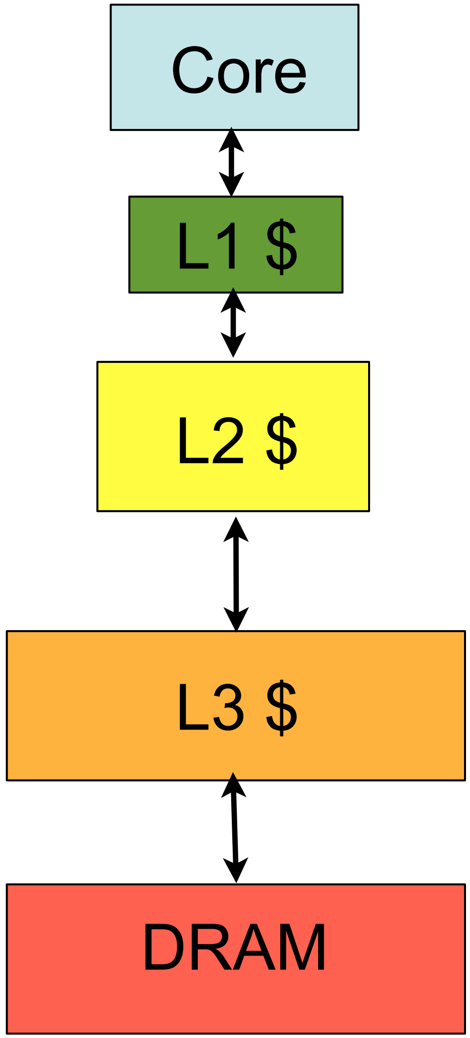
Memory Basics

- When data is not in registers, it needs to be fetched from somewhere (e.g. memory)
- Analogy:
  - the CPU is like a water mill
  - data is like the water
  - no matter how fast the CPU can run, if data is not flowing it will remain idle
- Memory fetches have
  - start up time (latency)
  - a ‘flow rate’ (B/W)



Caches

- If CPU is waiting for memory it sits idle
- memory fetches can take a long time (100s of CPU cycles)
- Hierarchy of “fast memories” added to store intermediate data
- These are called ‘caches’



1 of 4 cores, from Xeon X5570
2.933 GHz, 23.46 GFlops peak.

32K
Read Latency: 1.3ns (4cycles)
Read B/W: 45.6 GB/sec
Local to Core

256K
Read Latency: 3.4ns (10cycles)
Read B/W: 31.1 GB/sec
Local to Core

8M
Read Latency: 13.0ns (38 cycles)
Read B/W: 26.2 GB/sec
Shared between cores

Large: e.g 6x2GB DDR3
1333 MHz Dimms.
Read Latency: 65.1ns
(~190 cycles)
Read B/W: 10.1 GB/sec

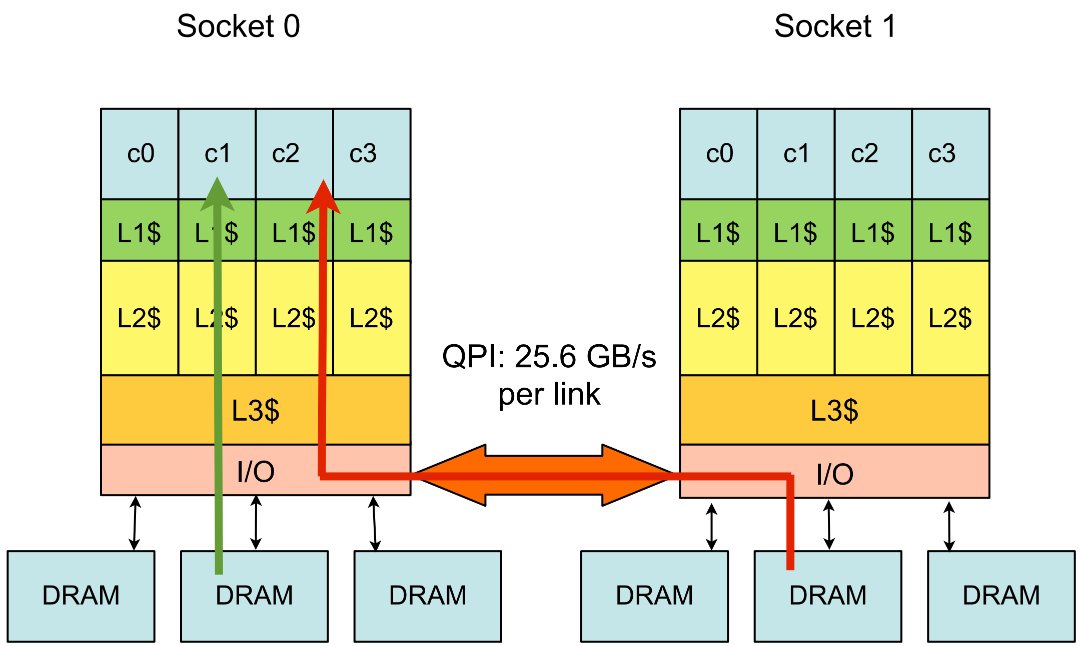
Numbers from: “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System” by D. Molka et. al., *2009 18th International Conference on Parallel Architectures and Compilation Techniques.* Latencies and bandwidths are from local cores only. Latencies tend to increase when reading from other cores. Multiple cores can draw more bandwidth. These numbers are for ***illustration*** only

Caches

- When the processor needs a data it will look in its cache first
  - if data is found (cache hit) it is fetched from cache
  - if not found (cache miss) a higher level of cache/or memory is tried.
- Caches work on the principle of locality
  - ***Spatial Locality:*** If I need a piece of data, its likely I will soon need another piece of data nearby in memory.
  - ***Temporal Locality:*** If I need a piece of data now, it is likely I may need it again soon.
- Caches process data in 'lines' containing multiple data values. (e.g. 64 bytes per line)

Multi-Core Complications

- Multiple latencies, BWs
  - from different cores
  - from different sockets
- Cache coherency
  - a core wants to write but who else has that data in their cache?
  - complicated protocols (snooping other cores' or sockets' caches)
- Collectively: Cache Coherent Non-Uniform Memory Access (ccNUMA)



Memory and TLBs

- CPUs typically operate using “virtual memory”
- Each process has its own ‘virtual address space’
  - as if the process owned all the memory
- In reality, multiple processes run share a physical memory
- CPUs have to translate virtual addresses from a process into physical addresses
- The TLB is a ‘cache’ to allow quick translation of addresses

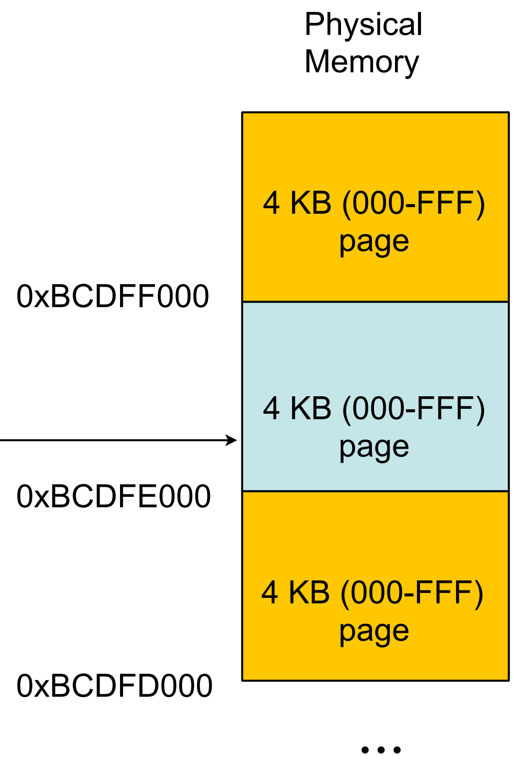
A TLB ‘hit’

Virtual Address: 0xABCDE010

TLB

Virtual address bits	Physical address bits
0xABCDE	0xBCDFE
0xABCDD	0x01223A
0x1263AB	0xFFFFA0
...	....

Translated Physical Address: 0xBCDFE010



- number of TLB entries is limited
- TLB misses are expensive (page walk)
- frequent large jumps (larger than page size) can cause frequent misses
- Often TLBs offer large pages (e.g. 2M)

Networks

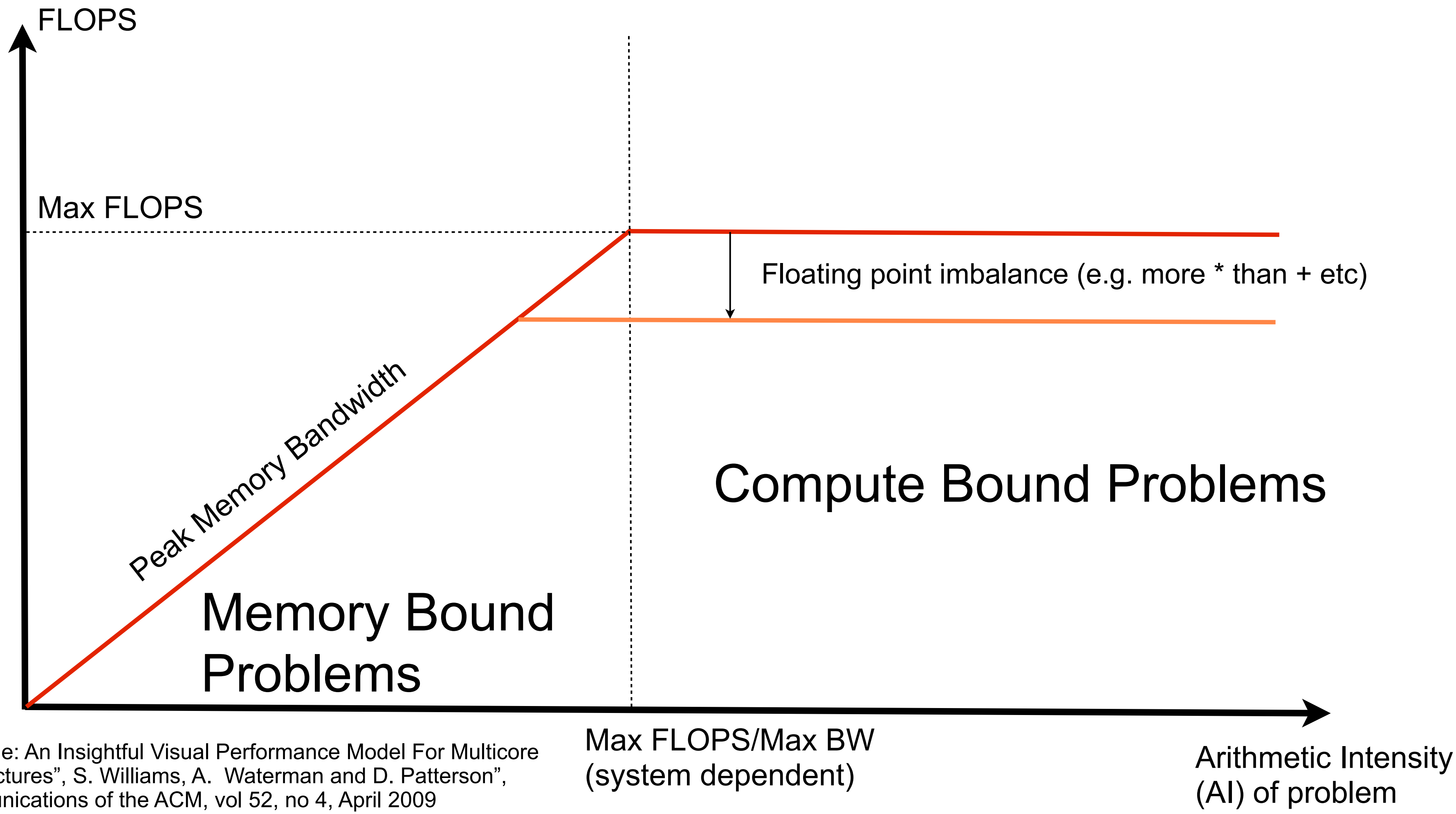
- Typically compute nodes are connected by several types of networks
  - Interconnect between Cores on a Socket (usually custom)
    - e.g. BG/Q crossbar
  - Inter Socket, Socket to Off Chip
    - e.g. Intel Quick Path Interconnect (QPI), AMD Hyper Transport (HT)
  - Other on node networks:
    - e.g. PCIe - to Graphics Processors or leading to Infiniband
  - Networks between compute nodes
    - Infiniband (commodity clusters),
    - Cray Gemini - connected with AMD processors via HyperTransport
    - BlueGene/Q - integrated onto the chip
- Like Memory: networks have latencies, and bandwidths
  - can sometimes just think of it also as remote memory...

Brief Recap

- Fundamentally we have resources we must manage within some constraints
  - Resources:
    - memory, cache, registers, vector units, cores, networks, accelerators
  - Constraints:
    - memory/network/cache latencies & bandwidths
    - size limits (# of registers, # of cache lines, # of TLB entries)
    - instruction issue limits (e.g. no of outstanding reads/writes, etc)
- Optimization is a process of balancing resources vs. constraints
  - Architect: balanced provision of resources within budget
  - Code: optimal use of relevant resources

Performance Limits: Roofline

- Arithmetic Intensity: Floating Point Ops/ Bytes of Data Used



“Roofline: An Insightful Visual Performance Model For Multicore Architectures”, S. Williams, A. Waterman and D. Patterson”, Communications of the ACM, vol 52, no 4, April 2009

Example: No reuse (streaming)

- AXPY: y[i] = a*x[i] + y[i], a is real, i=0...N-1
- 2 Flops for each element of x & y.
  - well balanced: 1 multiply, 1 add
  - need to load x[i] and y[i] for each 'i': 2 x 4 = 8 bytes
    - keep 'a' in a register
  - need to write out y[i]: another 4 bytes
  - Arithmetic Intensity: 2 FLOPS/12 bytes = 1/6
  - Speed of light for performance (working from memory)
    - on an Intel Core i7 3960X with mem b/w of 51.2 GB/sec: 8.53 Gflops
      - even tho the socket has a peak speed of ***316.8 Gflops***
    - if x & y fit into caches, ***higher cache B/W*** results in ***higher performance***
    - on an NVIDIA M2090 GPU with mem b/w of 177 GB/sec: 29.5 Gflops
      - even tho GPU can do ***1.3 Teraflops***

Example 2

- SU(3)xSU(3) matrix multiplication: M[i] = M1[i]*M2[i],   i=0..N-1
  - 108 multiplies, 90 adds for each value of 'i': 198 flops
  - 3x9 complex floats: 216 bytes
  - Arithmetic intensity: 198/216=0.92
- Maximum achievable performance (from Memory):
  - On system with mem B/W of 51.2 GB/sec: ~47 Gflops
- On a system with 316 GF peak and Mem B/W of 51.2 GB/s
  - need > 6.17 Flop/Byte to be compute bound (from DRAM)
- On a system with ~1360 GF peak and Mem B/W of 177 GB/s
  - need > 7.68 Flop/Byte to be compute bound (from GDDR)

What to take home from this

- If you can, run on enough nodes, so the local problem size fits in caches: then you are bound by cache bandwidth rather than main memory
  - but can have strong scaling issues elsewhere... (see later)
- If you are memory bandwidth bound, it means there are ‘free’ FLOPs. Use these where possible.

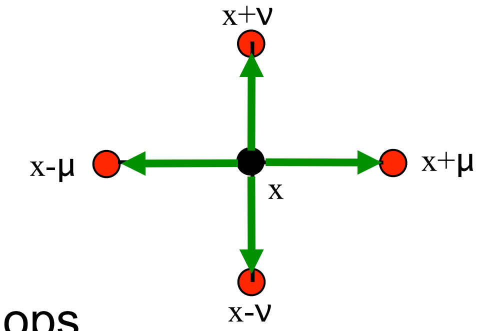
```
// AI: 1/6=2 flops/12 bytes
for(int i=0; i < N; i++){
   y[i]=a*x[i] + y[i];
}

// AI: 1/4=1 flop/4 bytes
double sum=0;
for(int i=0; i < N; i++) {
  sum+=y[i];
}
```

Loop Fusion

```
// AI: 1/4
// 12 bytes form axpy
// 3 flops
double sum=0;
for(int i=0; i < N; i++){
   y[i]=a*x[i] + y[i];
   sum +=y[i];
}
```

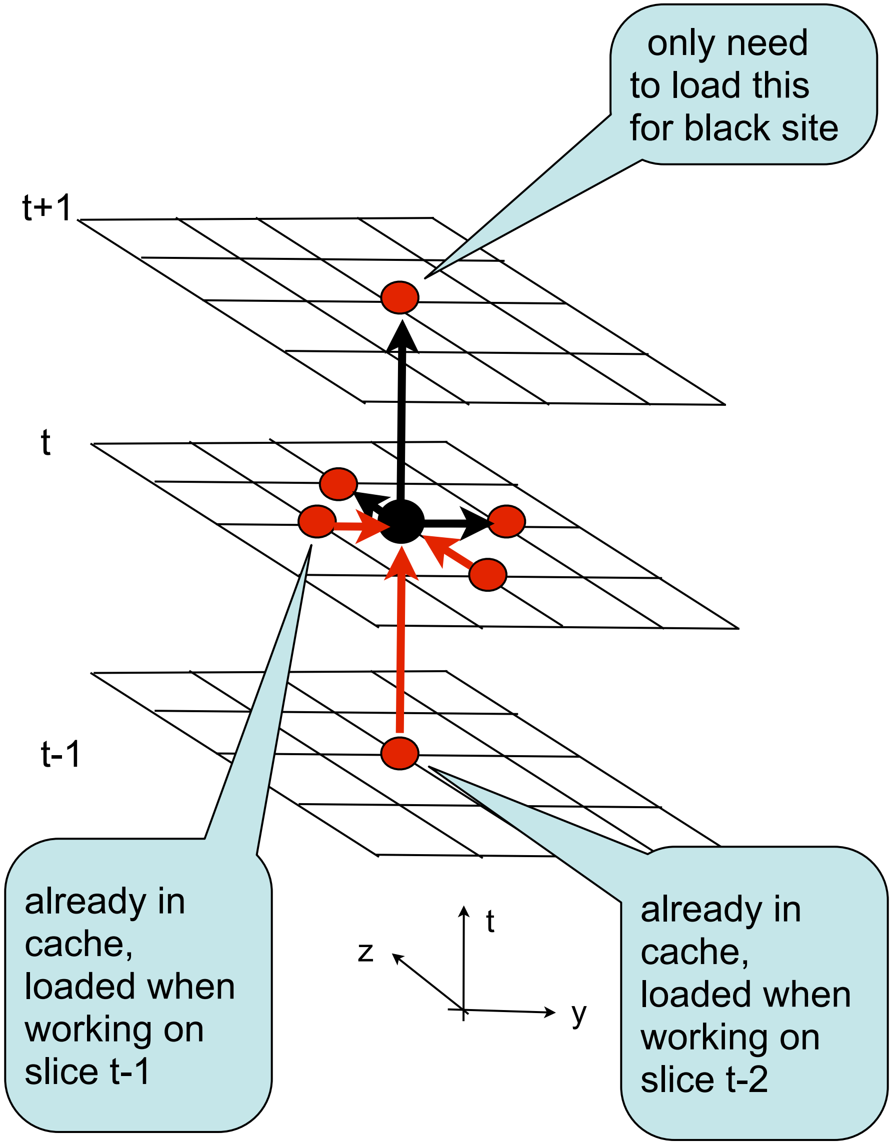
Case Study: Wilson Dslash



- We met Wilson Dslash in Lecture 2
- Naively: 1320 flops
  - For each of 8 directions (4 forward, 4 back)
    - SU(3) x color vector multiply for 2 spins: Total 8x2x66 flops
    - spin-projection: 8x12 flops,
    - spin reconstruct is ‘free’ (sign flips only)
  - Sum up 8 components => 7 summations: 7x24 flops
  - Total: 1320 flops
- Naive Bytes: 1440 bytes (single precision)
  - 8 gauge links (4 forward, 4 backward): 8x18x4 = 576 bytes
  - 8 input spinors (4 forward, 4 back): 8x24x4 = 768 bytes
  - 1 output spinor: 24x4=96 bytes
- Naive Flops/Bytes: **0.92 (Single Prec), 0.46 (double prec)**

Exploiting Spatial Reuse

- Consider 3D version (y,z,t plane)
  - ‘balls’ = spinors
  - ‘arrows’ = gauge links
- If cache is big enough
  - load input spinors for slice t, when working on t-1
  - load input spinor for slice t-1 when working on t-2
- 8-fold reuse of spinors (in 4D)
- no spatial reuse of gauge fields here
  - but for a 5D DWF type dslash, one can reuse gauge field L_5 times...



Dslash Perfomance Model

- If we can fit 3-time slices of spinors into a shared cache...
- Naive model:
  - Still 1320 FLOPs,
  - But only 768 bytes
    - still 576 for gauge, 96 for output spinor, but only 96 for the 1 spinor we load
  - Flops / Bytes ~ 1.72, much better than 0.92
- More sophisticated model (assume infinitely fast cache):

M. Smelyanskiy et. al.: "High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach", SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Article No. 69

$$\text{FLOPS=} \quad \frac{1320}{(576 + 96 + 96s)/R_{BW} + 96/W_{BW}}$$

- R_{BW}, W_{BW} are read/write bandwidths respectively
- s=0, if one has streaming stores, 1 otherwise
- For 2 dslashes in a row, there is also temporal reuse: see paper

Squeezing More from Memory

- SU(3) matrices allow several representations
  - 2 row representation, 8 real-number representation
  - reconstructing the full 3x3 matrix takes extra flops
  - but if we are memory bound, flops are 'free'
  - 2 row reconstruction: 42 flops/link, 336 flops/dslash
  - Arithmetic Intensity now:
    - Actual: 1656 flops/576 bytes = 2.875
    - Useful: 1320 flops/576 bytes = 2.29 (not counting the extra 336 flops)
  - See paper: ***Clark, et. al., Comp. Phys. Commun. 181:1517-1528, 2010***

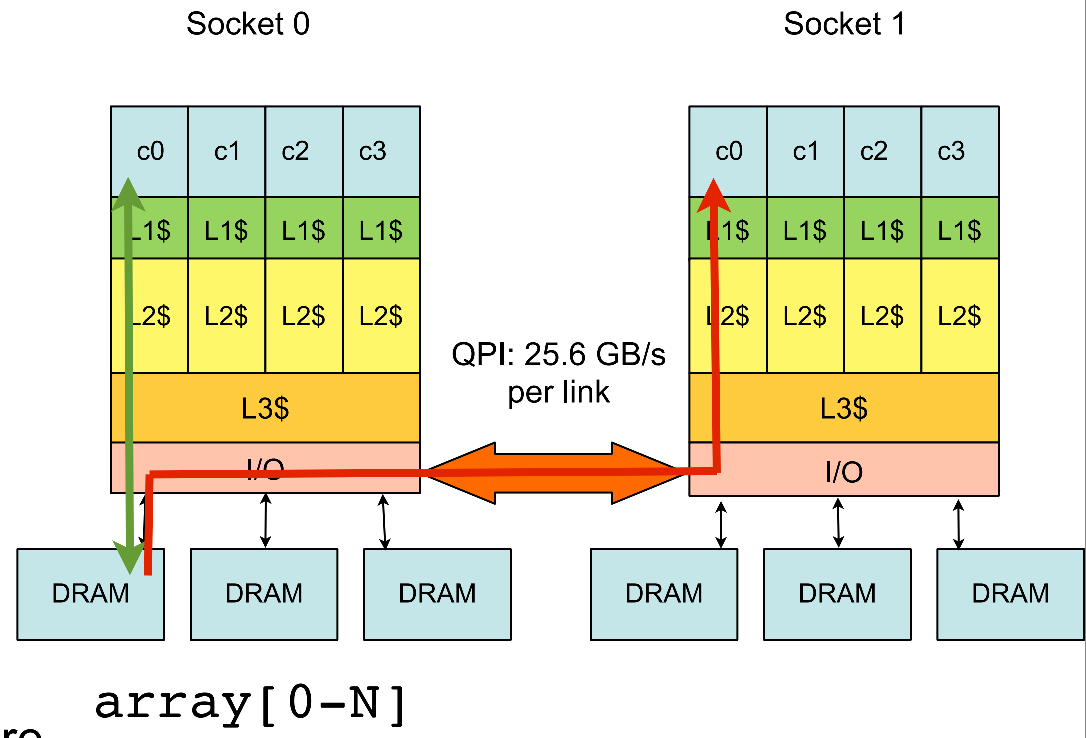
$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ \mathrm{x} & \mathrm{x} & \mathrm{x} \end{pmatrix} \xrightarrow{\begin{aligned} \mathbf{a} &= (a_1, a_2, a_3) \\ \mathbf{b} &= (b_1, b_2, b_3) \\ \mathbf{c} &= (\mathbf{a} \times \mathbf{b})^* \end{aligned}} \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix}$$

Inter Socket Communication

- NUMA and 'first touch'
  - the socket who writes memory first, 'owns' that memory.
  - e.g: this is not so good:

```
double array=new double[N];
for(i=0; i < N; i++)
  array[i]=drand48();

// sometime later on...
#pragma omp parallel for
for(i=0; i < N; i++) {
  array[i] *= 5*array[i];
}
```



  - master thread (e.g. socket 0, core 0) allocates and initializes 'array'
  - all worker threads get their array[i] from socket 0
  - QPI imposes bandwidth limit for cores on socket 1.

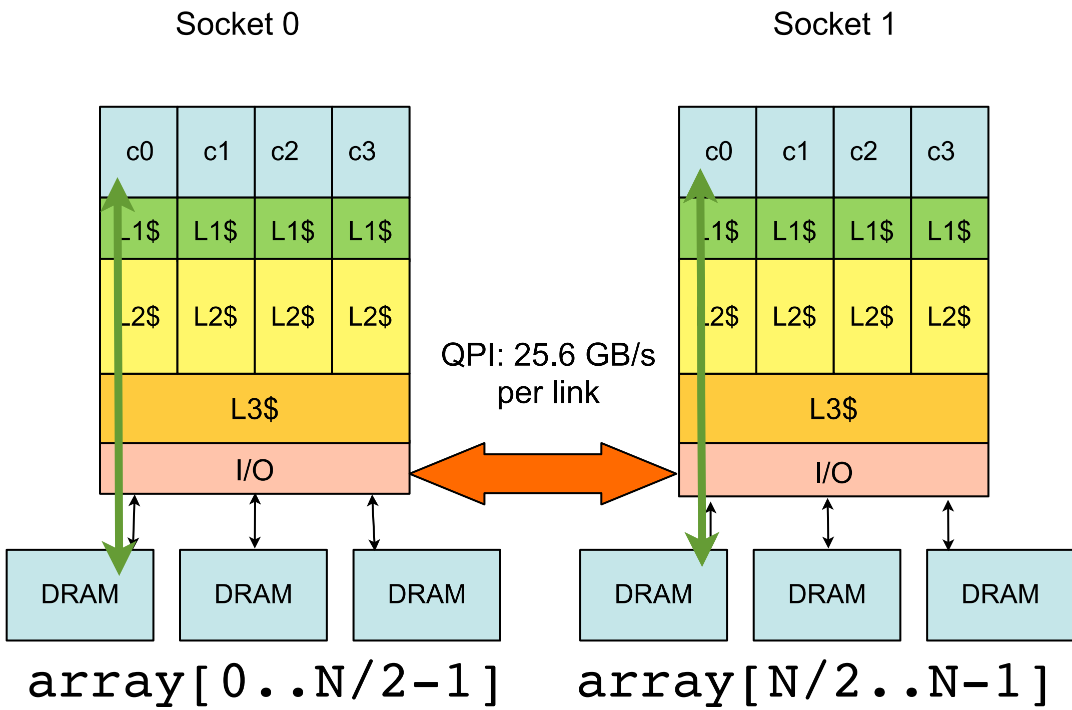
Inter Socket Communication

- Solutions:
  - use threaded loop to initialize array:

```
double array=new double[N];
#pragma omp parallel for
for(i=0; i < N; i++)
  array[i]=drand48();

// sometime later on...
#pragma omp parallel for
for(i=0; i < N; i++) {
  array[i] *= 5*array[i];
}
```

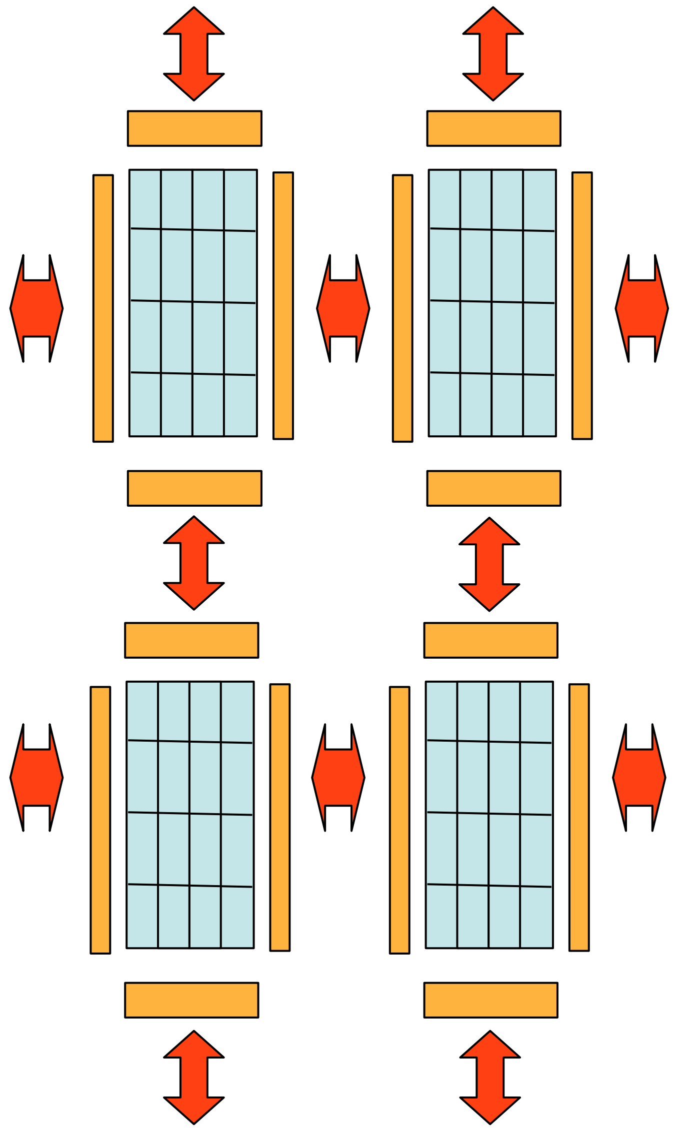
  - master thread (e.g. socket 0, core 0)
  - worker threads write to array to initialize
  - relevant parts of ‘array’ are ‘locally owned’



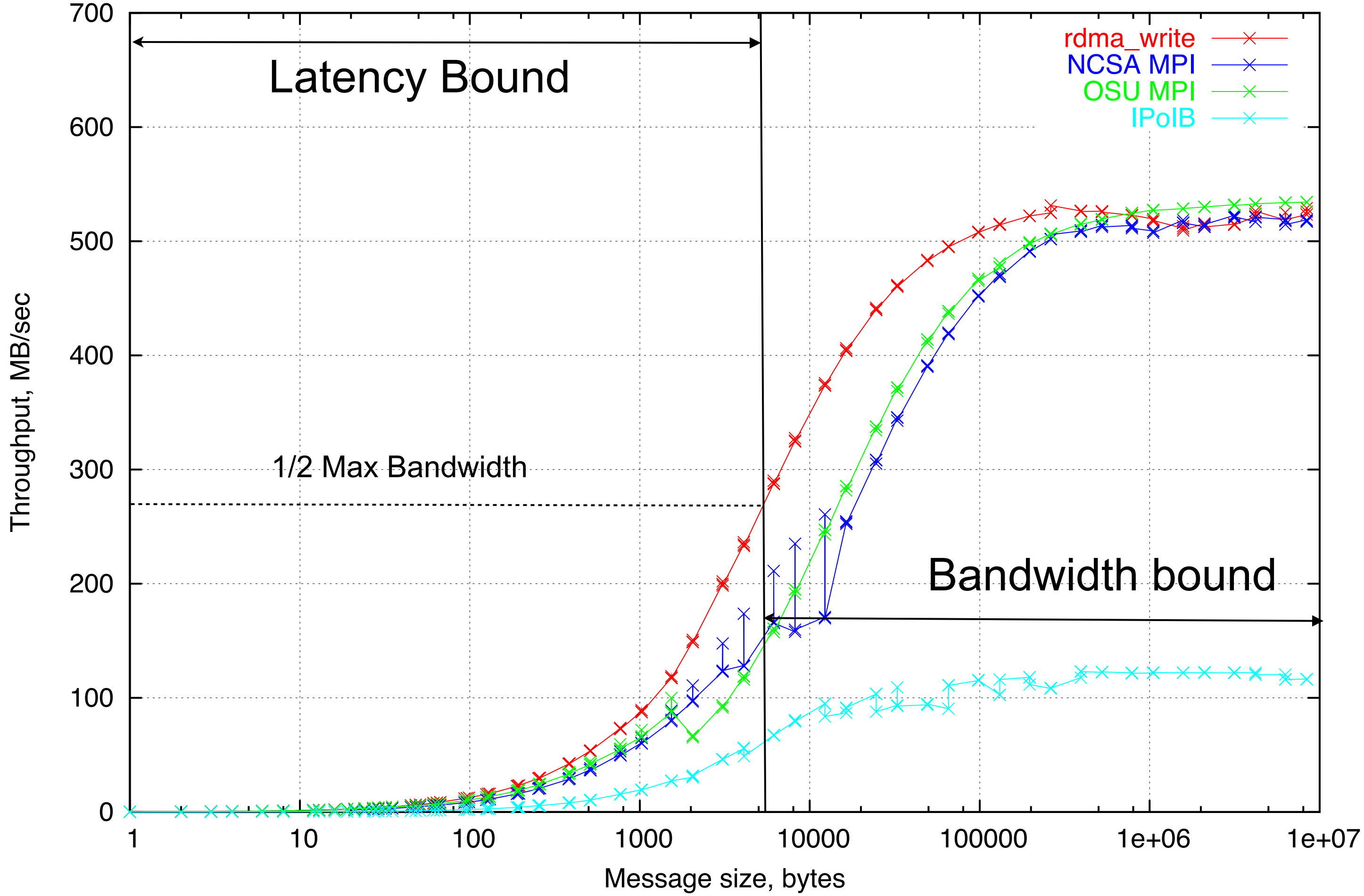
  - for this to work, threads must not migrate between sockets
  - alternative solution:
    - use one MPI Process per socket
    - all memory accesses ‘local’ unless sent via explicit messages.

Inter Node Communication

- Lattice QCD is local
  – mostly nearest/next-to-nearest neighbour communications.
- Need to communicate ‘faces’ of lattice
- This is a **surface** effect.
- Usually done via message passing through a network
  – network latency, bandwidth constraints
- Can work on local data while messages are ‘in flight’
- Overlap computation & communication



Messaging Characteristics

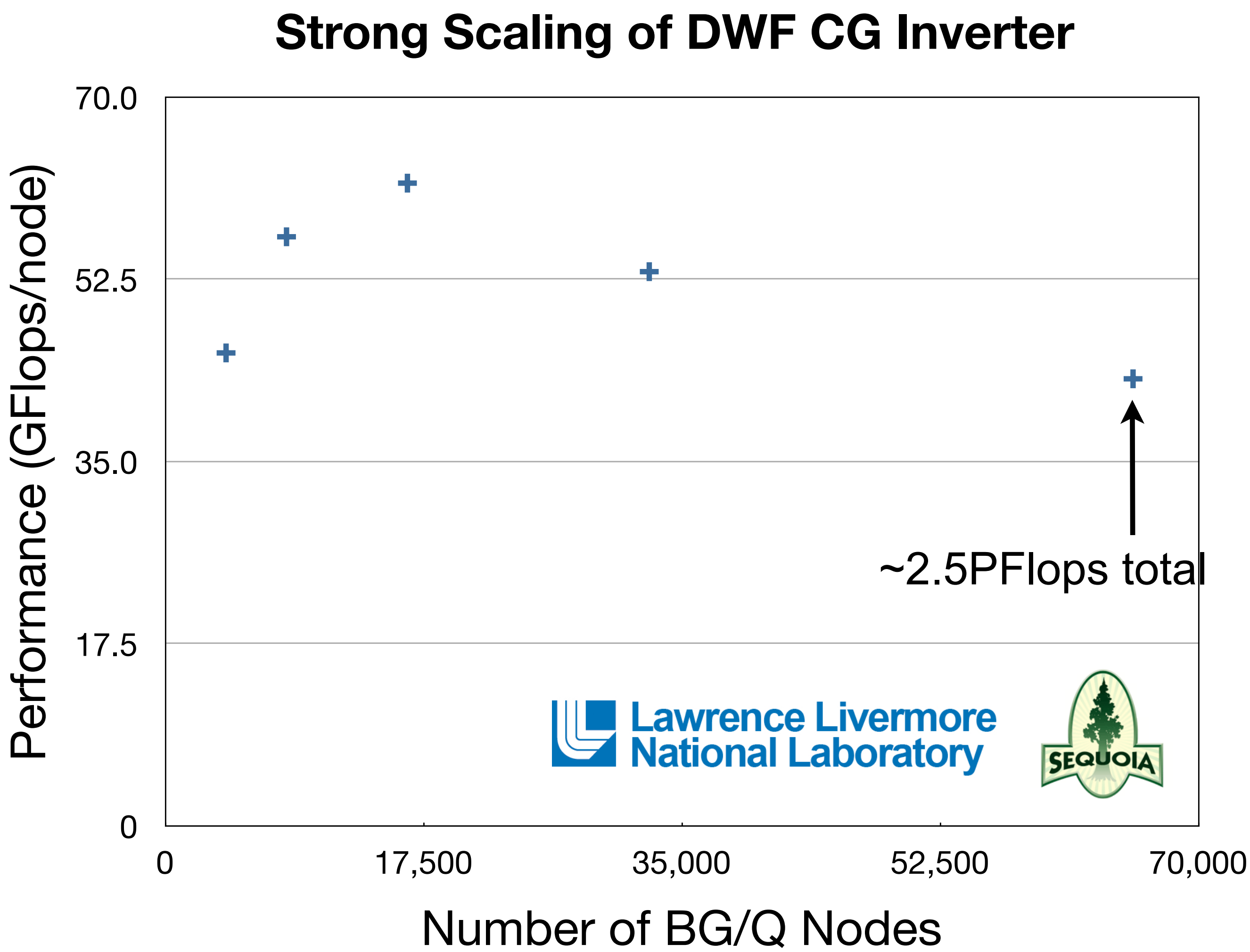


Performance of MPI over infiniband: http://lqcd.fnal.gov/benchmarks/newib/index.html

Optimality depends on situation

- Strong scaling regime
  - fix global volume, increase number of nodes
    - per node volumes become smaller
    - GOOD: We'll fit into caches better.
    - but also BAD: Surface to volume ratio gets worse
    - Makes overlapping computation w. communication more difficult
      - e.g. $4x2^3$ local lattice, 2^4 after checkerboarding, all surface, no 'body'
      - Small messages become latency bound, need low latency interconnects
- Larger volumes, fewer nodes (e.g. a cluster)
  - large volumes per node
    - Surface to volume small: Bandwidth Bound
    - More local data: less likely to fit in cache, need more memory bandwidth
    - Fewer nodes: need more powerful nodes (e.g. GPU accelerated)
  - Optimal choice of hardware depends on many factors
    - e.g. FLOP/$, FLOP/W & W/$, but $$$$ always in there somewhere.

BG/Q is designed to strong scale



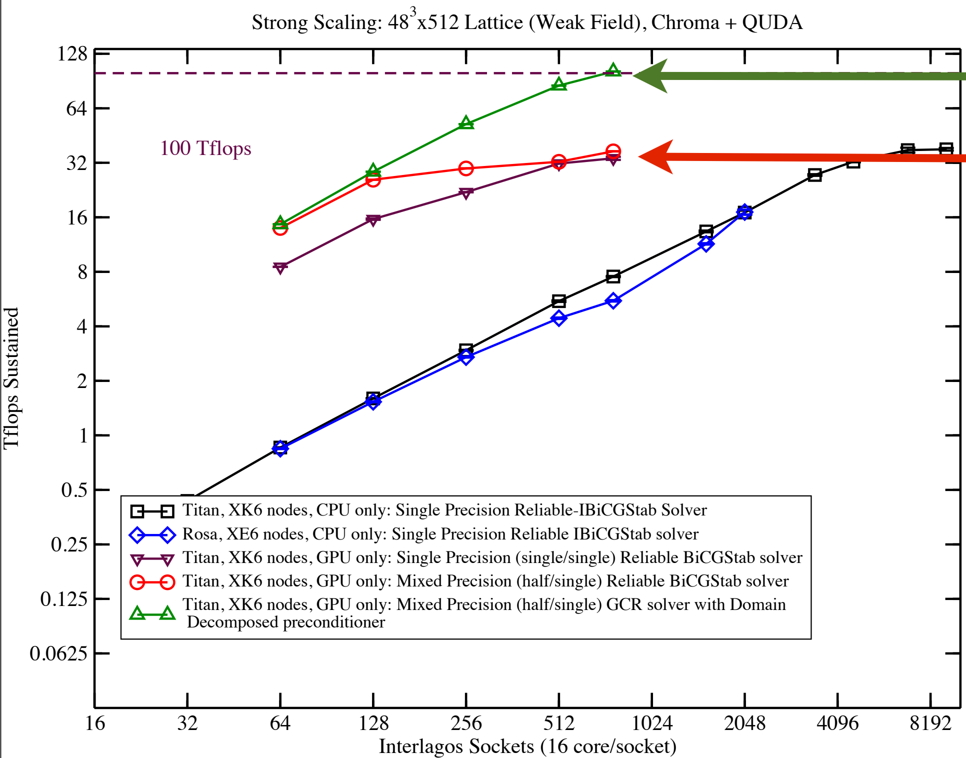
On BG/Q messaging unit and network are integrated on-chip.

Figure courtesy of M. Buchoff (LLNL)

Code developed by Peter Boyle at the STFC funded DiRAC facility at Edinburgh

Optimization: Communicate less

- Strong scaling can be difficult for GPU systems
  - PCIe-2 bus bandwidth (8+8 GB/s peak, 5+5 GB/s in practice)
  - multiple-hops: GPU to CPU to Network to CPU to GPU (high latency)
  - situation is getting better: PCIe-3 is coming, GPUDirect reduces latency
  - can one strong scale in the interim? Yes: use reduced communication algorithm



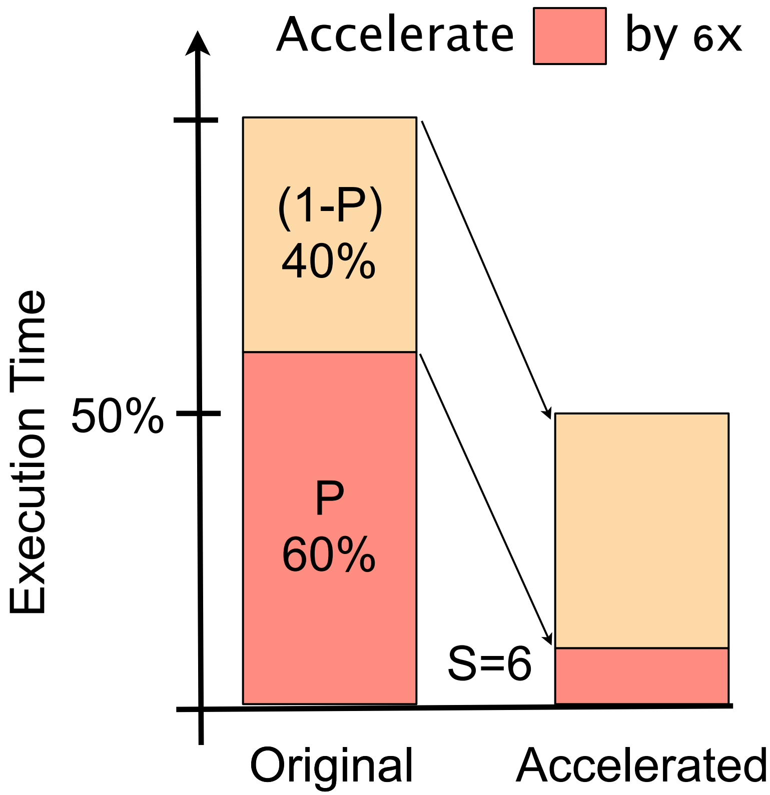
DD+GCR: reduced communications algorithm

BiCGStab, regular communications

- DD+GCR
  - GCR solver with Block Diagonal preconditioner
  - Preconditioner does no comms.
  - scaling now limited by
    - comms in outer GCR process
    - GPU performance at small local volumes

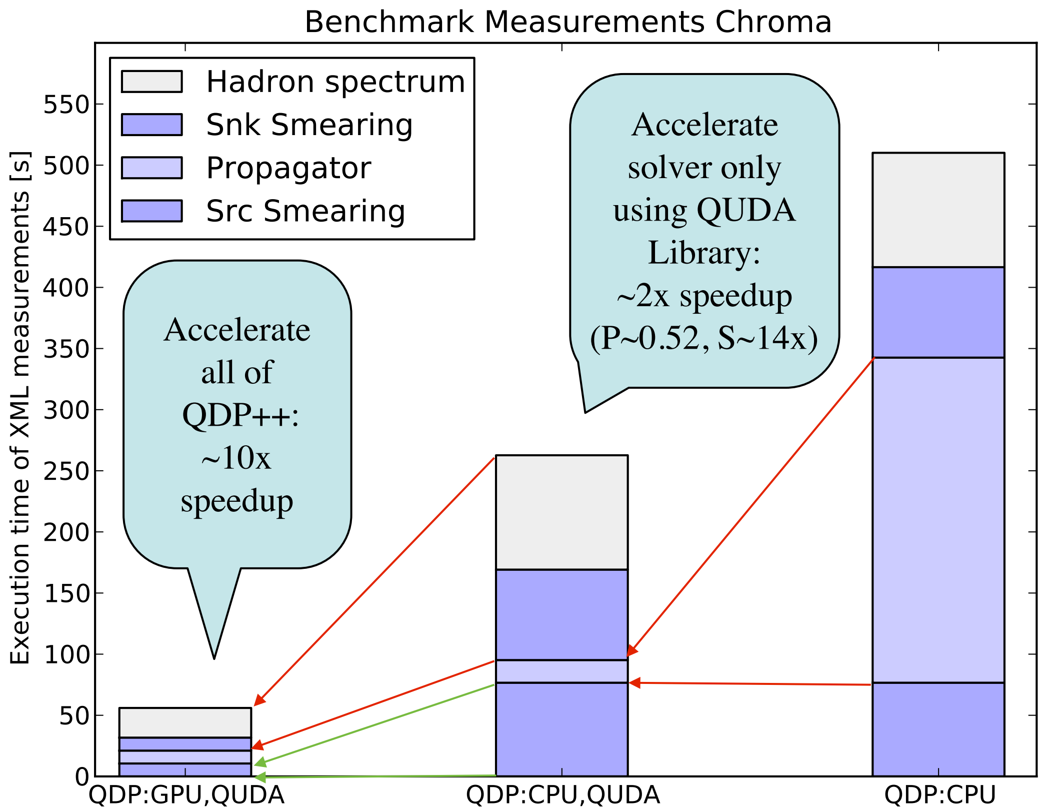
Amdahl's Law

$$S_{\mathrm{app}} = \frac{1}{(1-P) + \frac{P}{S}}$$



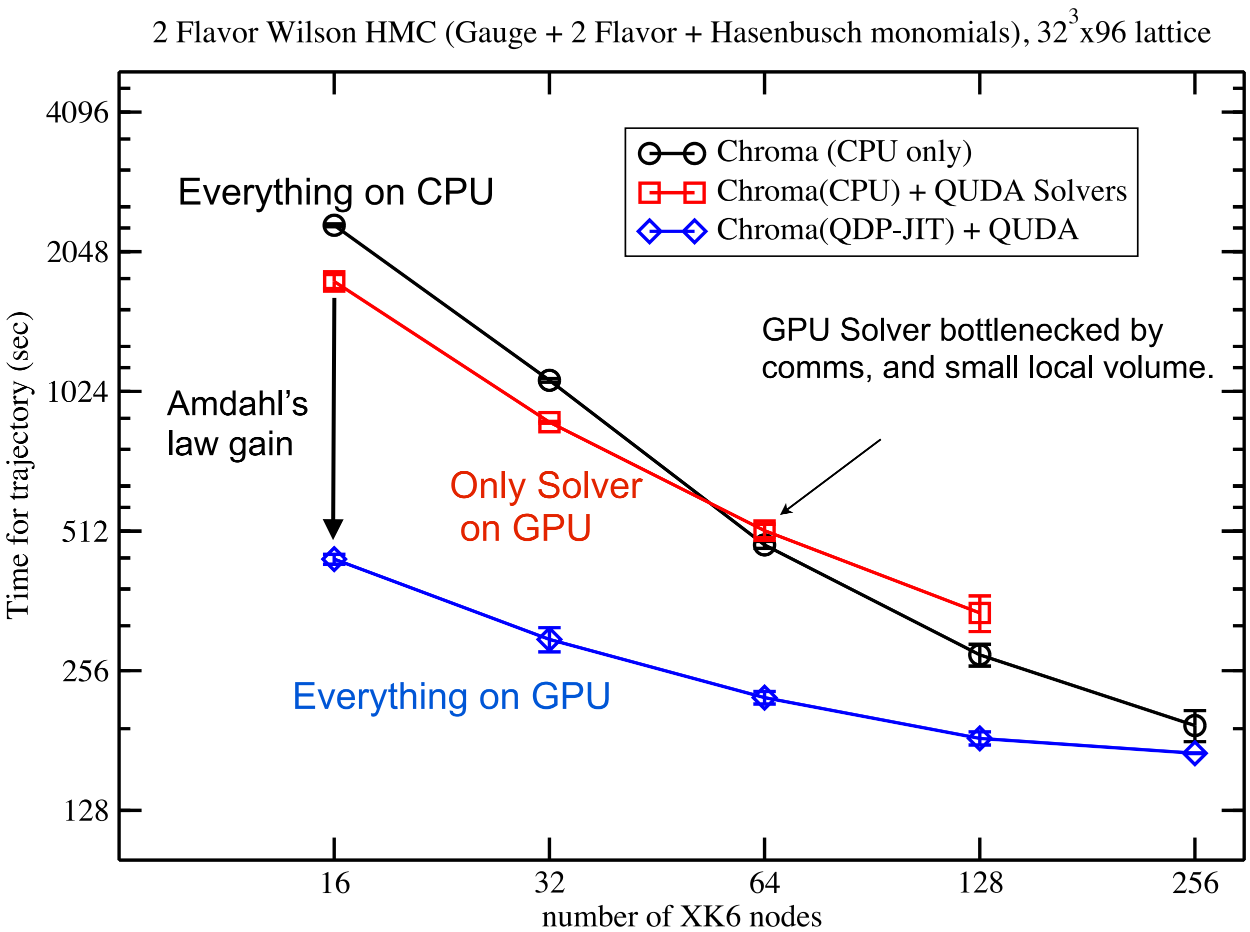
- Puts optimization into perspective.
  - Speed up part of code where proportion 'P' of time is spent by a factor 'S'
  - Overall execution is S_{app} faster
- ie: Optimize where it matters!
- Accelerate 60% of code by 6x and your overall speed up is 2x
- Amdahl's law can also be applied to any other form of speed increase to a portion of the code
  - using more processors, accelerators etc.
- Rule: increase P if you can.

Beating Down Amdahl's Law on GPUs



- Results from Frank Winter's talk at Autumn StrongNET meeting (Trento, 2011)
- QUDA alone only gave ~2x speedup on full application
- QUDA + moving all of QDP++ to GPU resulted in ~10x speedup
- See also: F. Winter "Accelerating QDP++ using GPUs" arXiv:1105:2279[hep-lat]

Works also for Gauge Generation



PRELIMINARY

Messages to take away

- You should be systematic about your optimization
  - measure where your code spends time
  - identify which parts you want to speed up
  - consider the kind of optimization, the effort and payback
    - consider performance limits, work with a perf model if you have one
    - consider algorithmic improvement, rather than just performance improvement (work smarter, not just harder/faster)
    - Consider the effort involved. Would you have finished with the original code by the time you make the improvements? Will the improvements benefit you later on?
    - Consider whole application performance improvements (Amdahl's law). How much overall improvement will your specific optimization bring.

Conclusions

- The ideas here are generic, and should be transferable
- Ideas used in GPUs not always so different from CPUs

GPU	CPU
coalesced reads	cache-line reads
block for shared memory	block for cache
manage grid of thread blocks	manage thread placement/binding
host/device data movement	message passing, NUMA 1st touch
shared memory bank conflict	cache associativity conflict
register blocking	SIMD vectorization

- The hard part is figuring out how to exploit the resources you have and how you will deal with the constraints & bottlenecks
- The rest is just typing... -- but lots of typing...
- Optimization and performance tuning can be all consuming
  – but sometimes a lot of fun :)