Graph Analytics with lots of CPU cores

Tim Mattson
(timothy.g.mattson@intel.com)
Intel, Parallel Computing Lab

Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
-

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Disclaimer

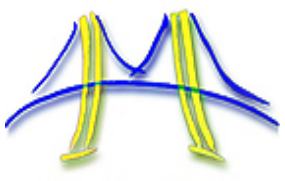
- The views expressed in this talk are those of the speaker and not his employer.
- If I say something "smart" or worthwhile:
  - Credit goes to the many smart people I work with.
- If I say something stupid…
  - It's my own fault

I work in Intel's research labs. I don't build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don't miss any great ideas.

Hence, my views are by design far "off the roadmap".

Acknowledgments:

- Many people have contributed to these slides.



  - Slides produced in partnership with Kurt Keutzer and his group at UC Berkeley are marked with the UCB ParLab logo

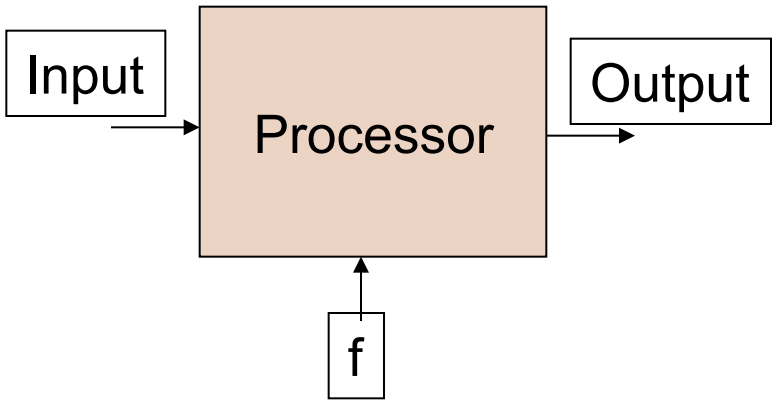
- Slides from Aydin Buluc (LBL) and John Gilbert (UCSB) include the UCSB logo. Many of these slides also had input from Jeremy Kepner (MIT) and David Bader (Georgia Tech).

- And too many Intel people to list … the graphMAT team (Nadathur Rajagopalan Satish, Narayanan Sundaram, Mostofa Ali Patwary, Vadlamudi Satya Gautam and Michael Anderson). Victor Lee for his "debunking the GPU myth" slides, Pradeep Dubey for Machine learning input and the excellent Hot chips 2015 talk by Avinash Sodani

Outline

- The move to many core processors
- Why Linear Algebra is at the heart of data analytics
- Working with Many core CPUs
- Accelerators and the future of hardware
- A programmer's response to Hardware changes.

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$C = q/V \rightarrow \quad q = CV$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$V * q = W.$

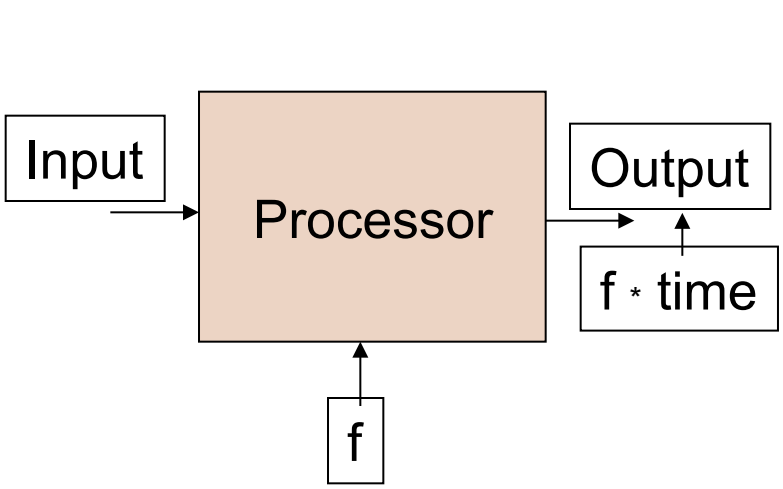
But for a circuit $q = CV$ so

$W = CV^2$

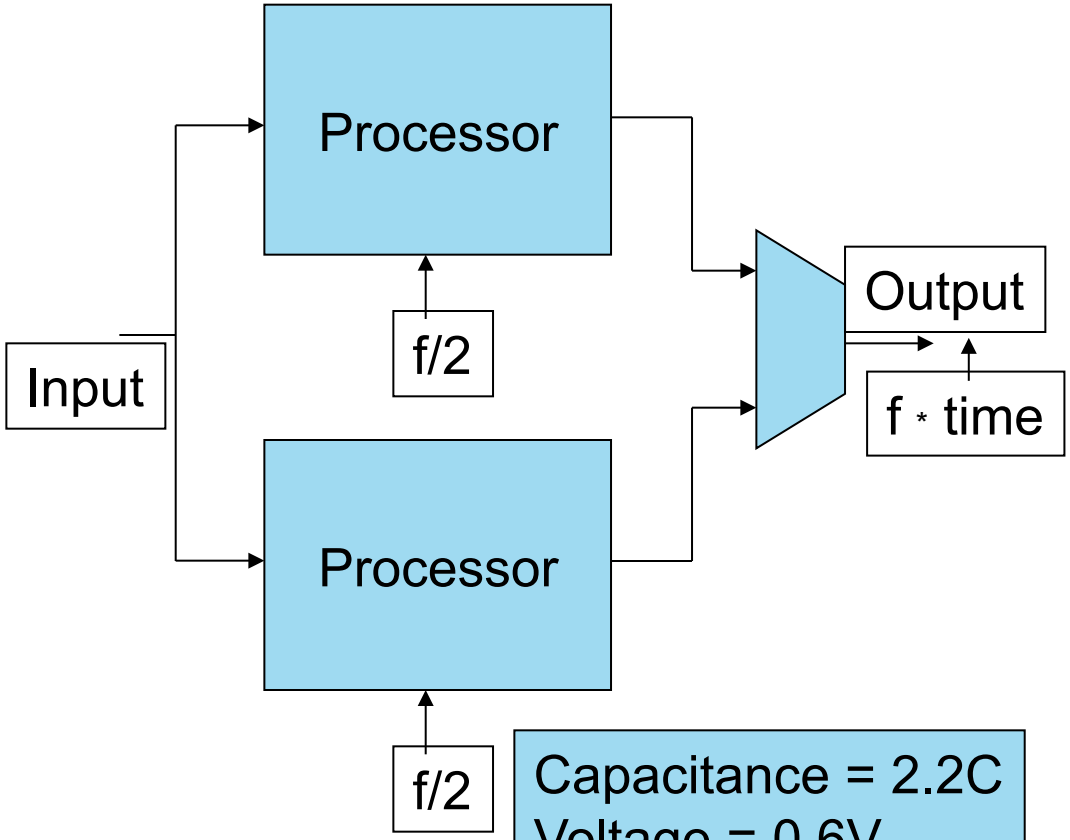
power is work over time … or how many times in a second we oscillate the circuit

Power = W* F → Power = CV^2f

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

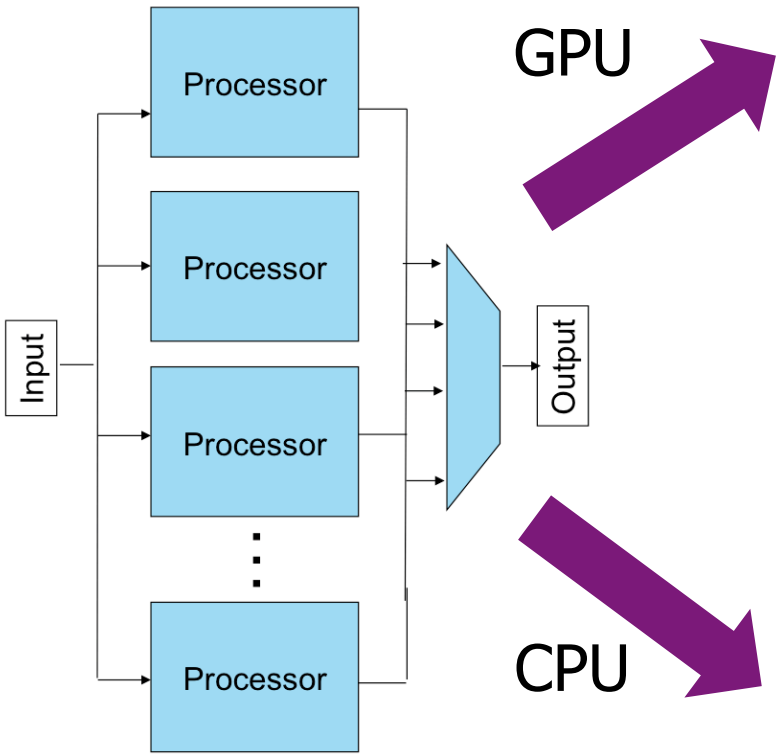


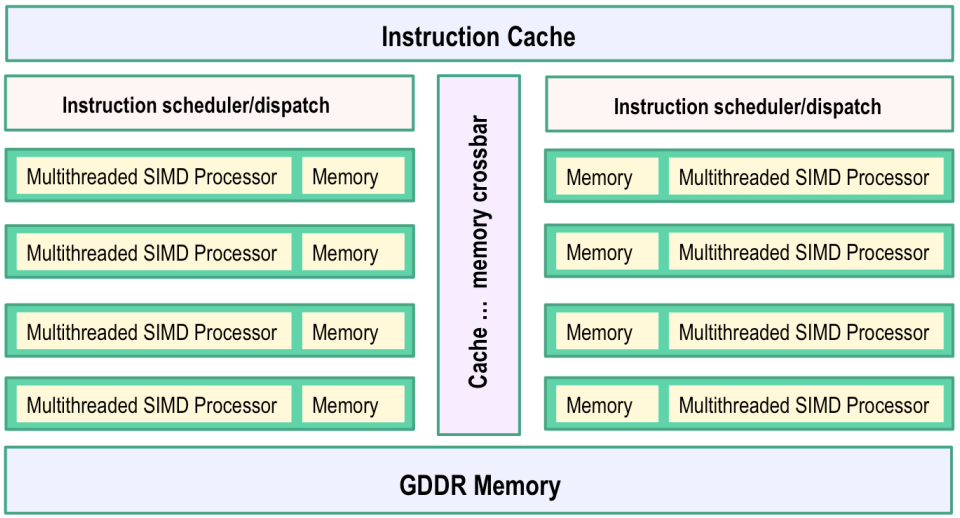
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

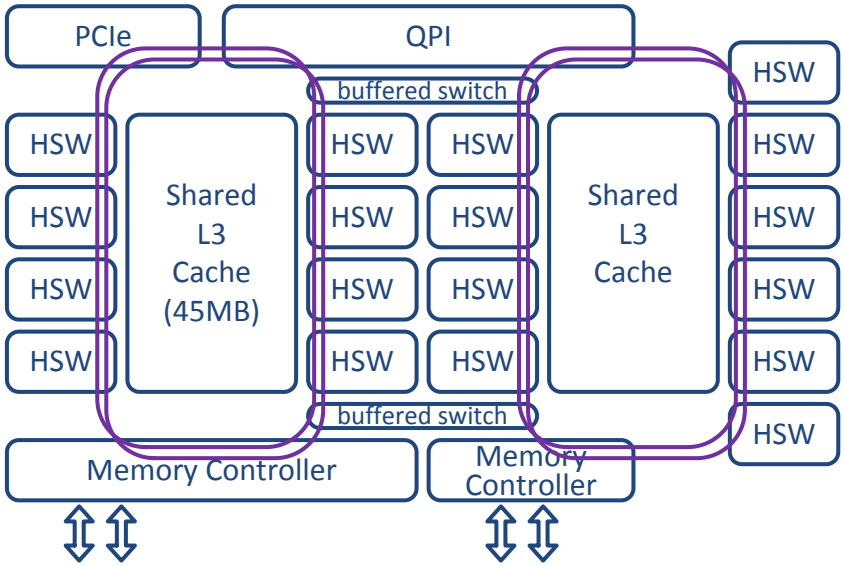
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

Source:
Vishwani Agrawal

... Many core: we are all doing it

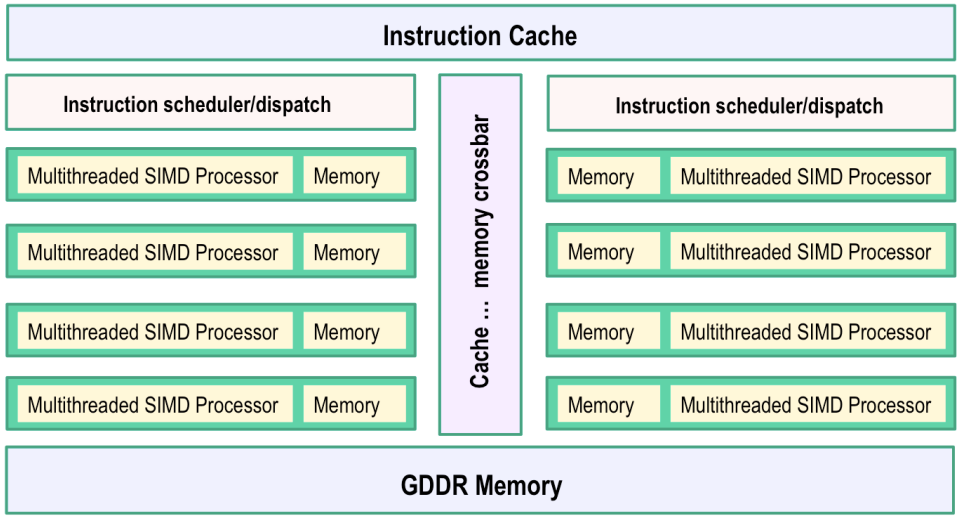






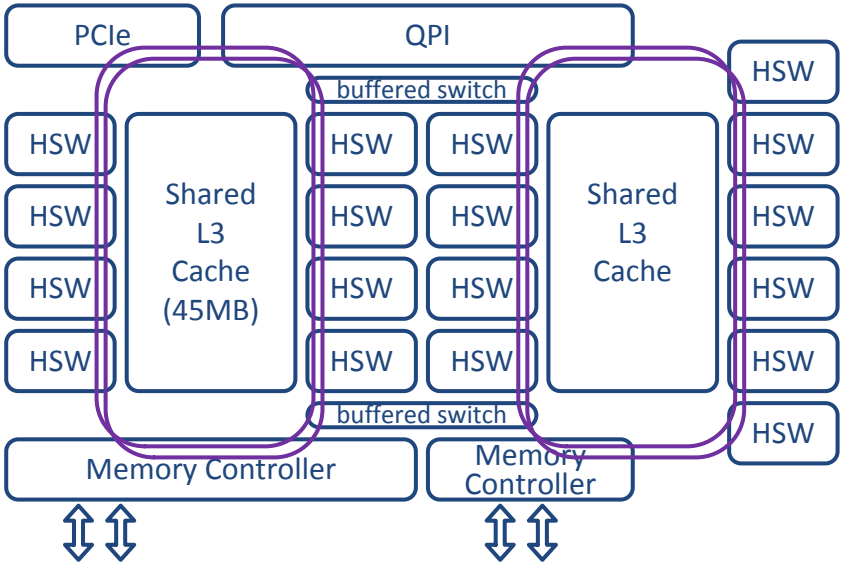
"HSW" refers to a particular type of core (i.e. a "Haswell" core).

... Many core: we are all doing it



GPU

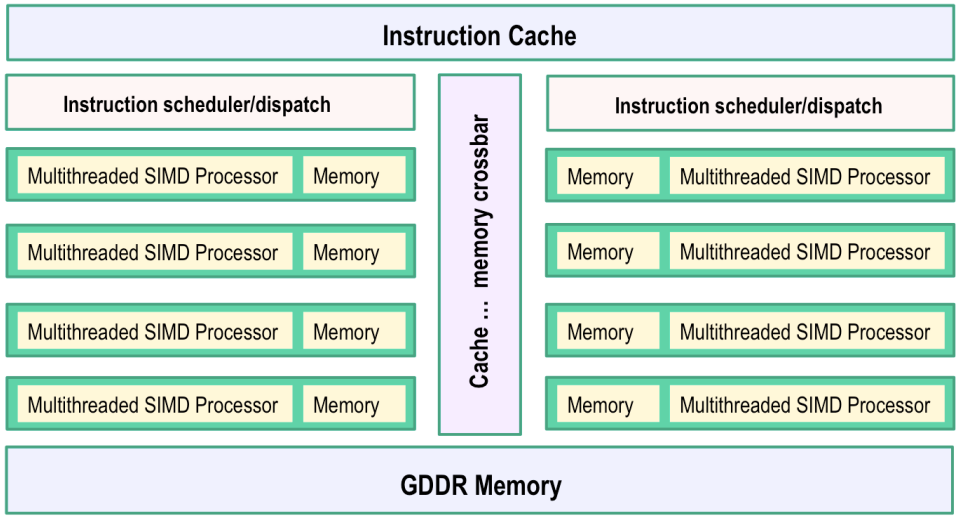
- Hundreds of Cores Driven by a single stream of instruction
- Each core has many (32, 64, ...) SIMD lanes
- Optimized for throughput ... oversubscription to hide memory latency



CPU

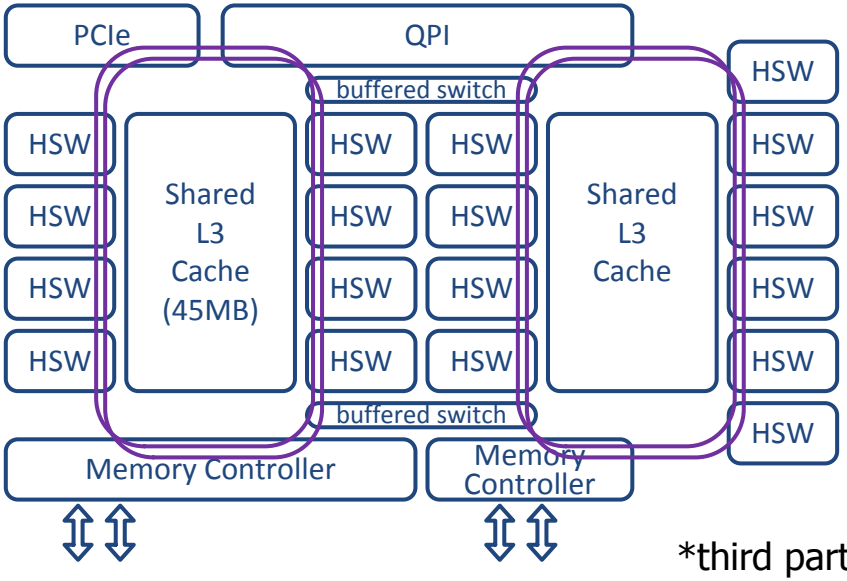
- A few to dozens of Cores with independent streams of instructions.
- Cores typically have complex logic (e.g. out of order) to make individual threads run fast.
- Optimized for latency ... complex memory hierarchy to run fast out of cache

It's really about competing software platforms



GPU

- **Single Instruction multiple threads.**
  - turn loop bodies into kernels.
  - HW intelligently schedules kernels to hide latencies.
- *Dogma*: a natural way to express huge amounts of data parallelism
- Examples: CUDA, OpenCL, OpenACC



CPU

- **Shared Address space, multi-threading.**
  - Many threads executing with coherent shared memory.
- *Dogma*: The legacy programming model people already know. Easier than alternatives.
- Examples: OpenMP, Pthreads, C++11

*third party names are the property of their owners

Outline

- The move to many core processors
- Why Linear Algebra is at the heart of data analytics
- Working with Many core CPUs
- Accelerators and the future of hardware
- A programmer's response to Hardware changes.

Motivation: History of BLAS

- BLAS: The Basic Linear Algebra subroutines

BLAS 1		Lawson, Hanson, Kincaid and Krogh, 1979	LINPACK
BLAS 2		Dongarra, Du Croz, Hammarling and Hanson, 1988	LINPACK on vector machines
BLAS 3		Dongarra, Du Croz, Hammarling and Hanson, 1990	LAPACK on cache based machines

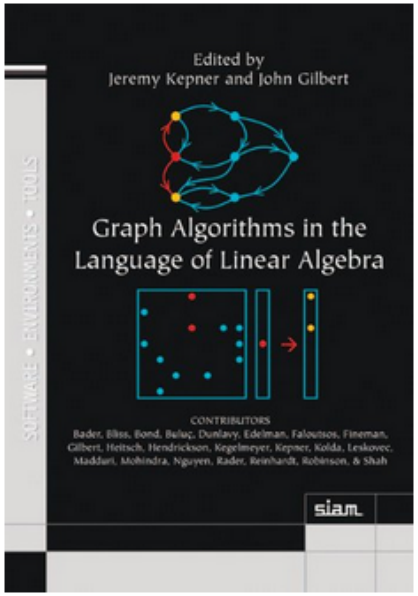
- The BLAS supported a separation of concerns:
  - HW/SW optimization experts tuned the BLAS for specific platforms.
  - Linear algebra experts built software on top of the BLAS .. high performance “for free”.

- It is difficult to overestimate the impact of the BLAS … they revolutionized the practice of computational linear algebra.

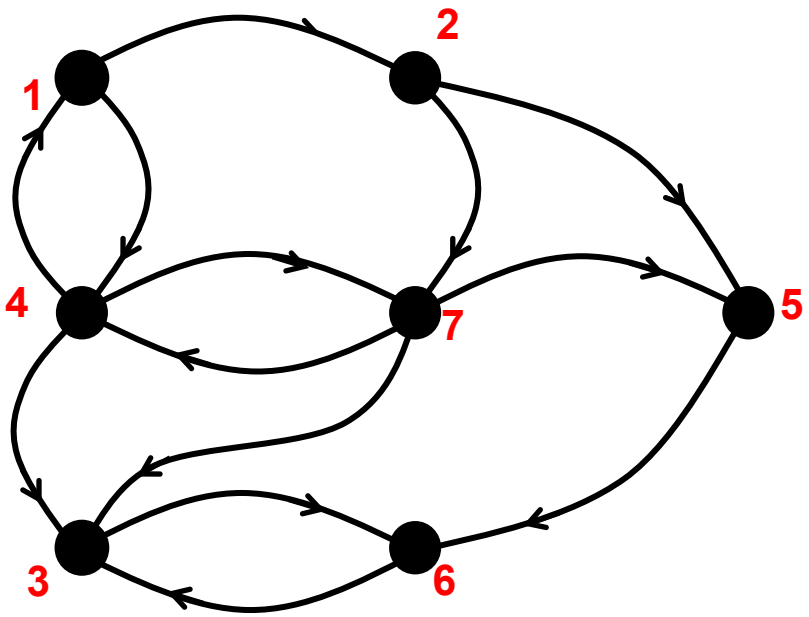
Can we standardize "the BLAS" of graph algorithms

- No, it is not reasonable to define a common set of graph algorithm building blocks:
  - Matching Algorithms to the hardware platform results in too much diversity to support a common set of "graph BLAS".
  - There is little agreement on how to represent graph algorithms and data structures.
  - Early standardization can inhibit innovation by locking in a sub-optimum status quo

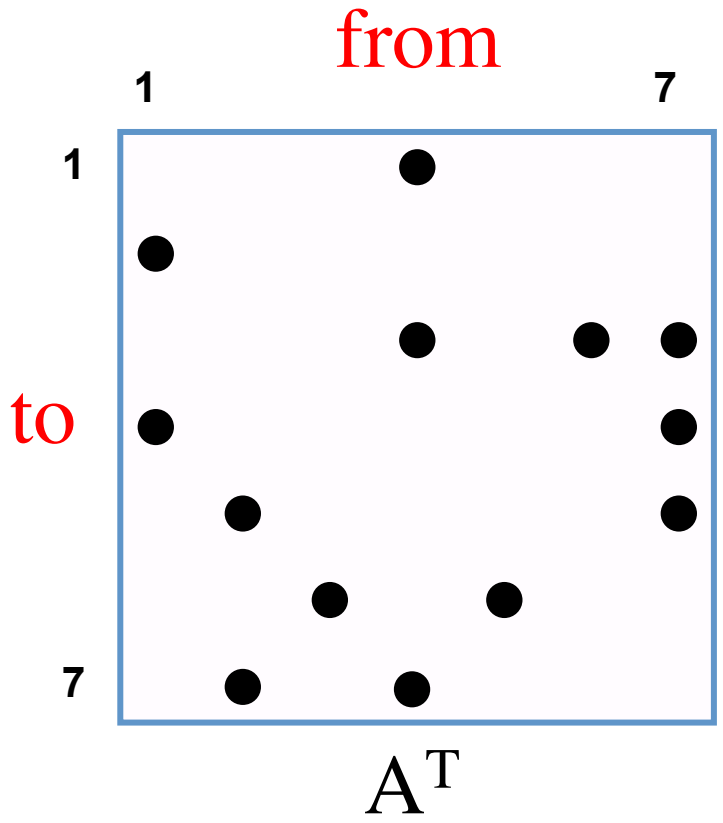
- Yes, it is reasonable to define a common set of graph algorithm building blocks … for Graphs in the language of Linear algebra.
  - Representing graphs in the language of linear algebra is a mature field … the algorithms, high level interfaces, and implementations vary, but the core primitives are well established .



Graphs in the Language of Linear Algebra

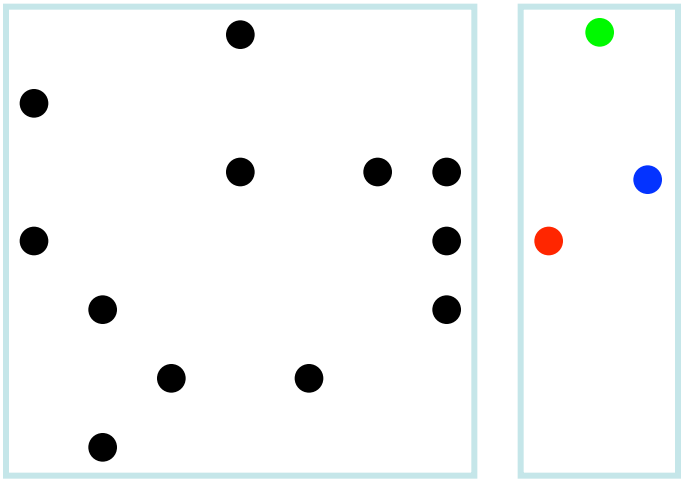


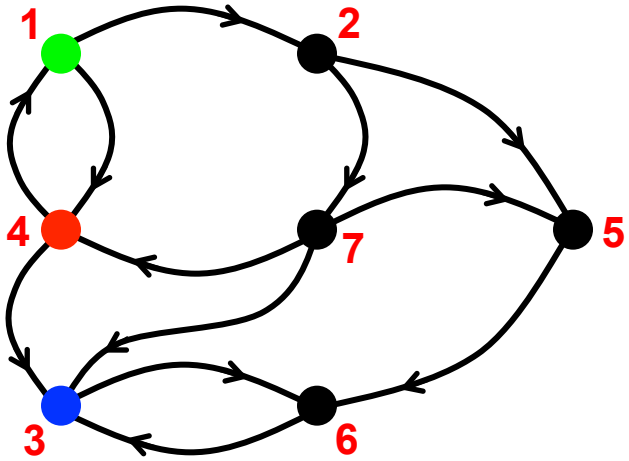
These two diagrams are equivalent representations of a graph.



A = the adjacency matrix … Elements nonzero when vertices are adjacent

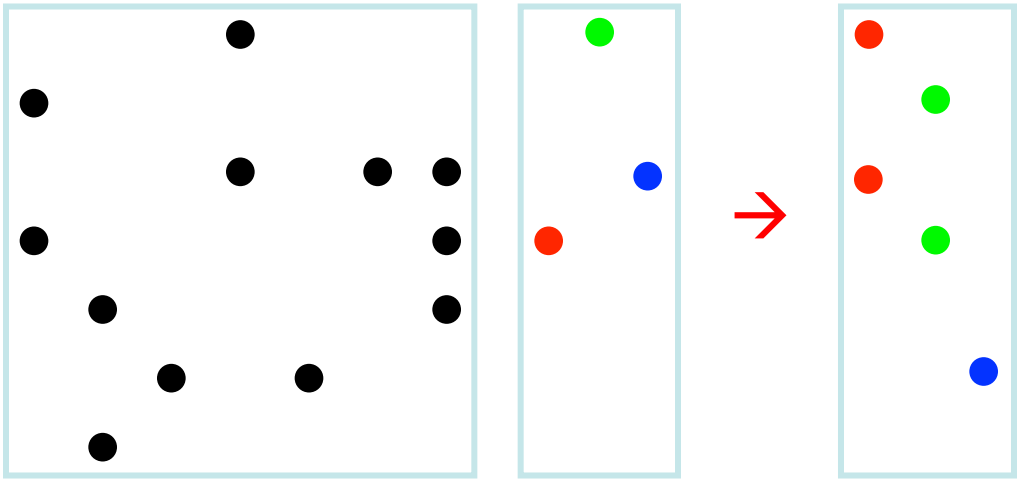
Multiple-source breadth-first search

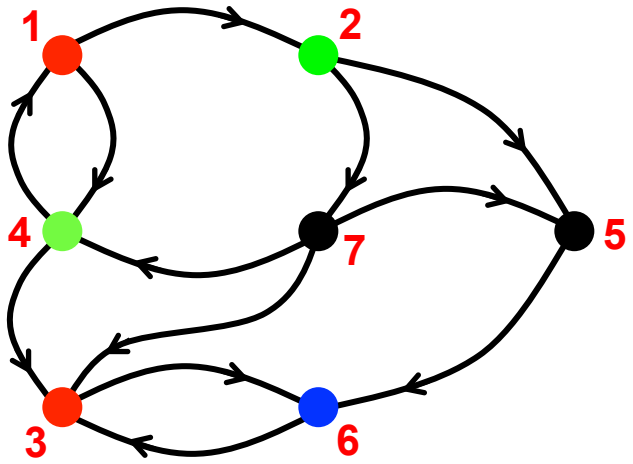




UCSB

Multiple-source breadth-first search





- Sparse array representation => space efficient
- Sparse matrix-matrix multiplication => work efficient
- Three possible levels of parallelism: searches, vertices, edges

Multiplication of sparse matrices captures Breadth first search and serves as the foundation of all algorithms based on BFS

UCSB

Moving beyond BFS with Algebraic Semirings

- A semiring generalizes the operations of traditional linear algebra by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

Moving beyond BFS with Algebraic Semirings

- A semiring generalizes the operations of traditional linear algebra by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

(R, +, *, 0, 1) Real Field	Standard operations in linear algebra

Notation: (R, +, *, 0, 1)

Scalar type — Op1 — Op2 — Identity Op1 — Identity Op2

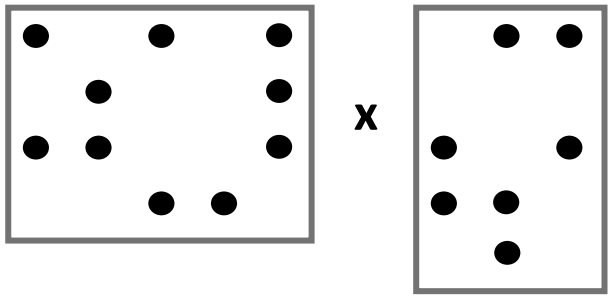
Moving beyond BFS with Algebraic Semirings

- A semiring generalizes the operations of traditional linear algebra by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

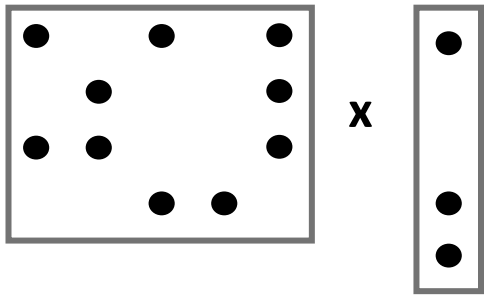
(R, +, *, 0, 1) Real Field	Standard operations in linear algebra
({0,1}, |, &, 0, 1) Boolean Semiring	Graph traversal algorithms
(R U {∞},min, +, ∞, 0) Tropical semiring	Shortest path algorithms
(R U {∞}, min, *, ∞, 1)	Selecting a subgraph or contracting nodes to form a quotient graph.

Linear-algebraic primitives

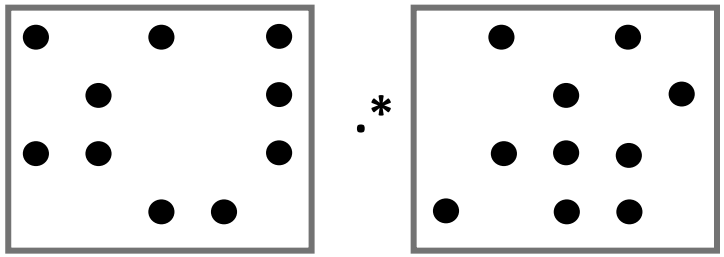
Sparse matrix-sparse matrix multiplication



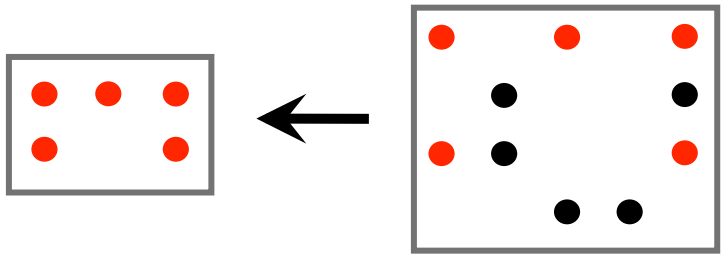
Sparse matrix-sparse vector multiplication



Element-wise operations



Sparse matrix indexing



The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. (×, +), (and, or), (+, min)

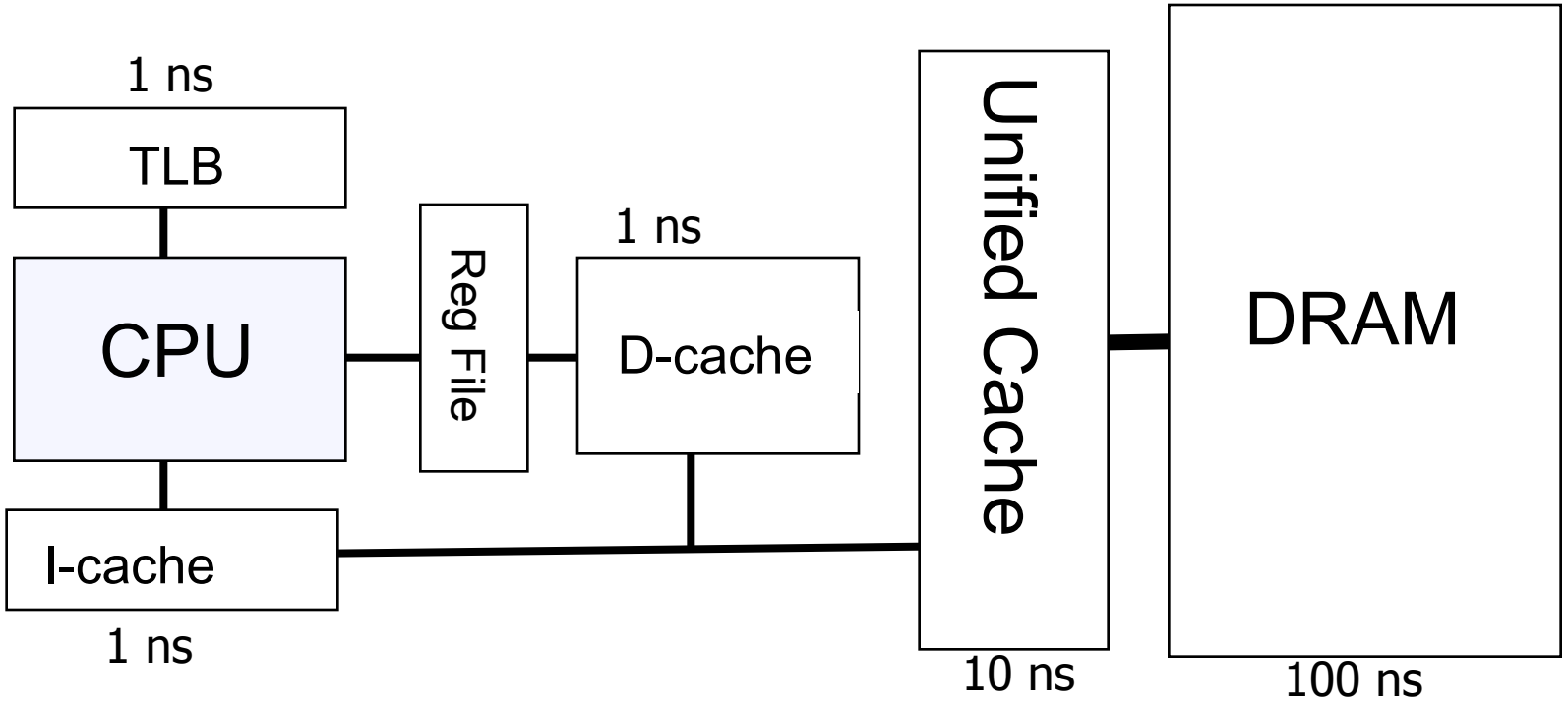
UCSB

Outline

- The move to many core processors
- Why Linear Algebra is at the heart of data analytics
- Working with Many core CPUs
- Accelerators and the future of hardware
- A programmer's response to Hardware changes.

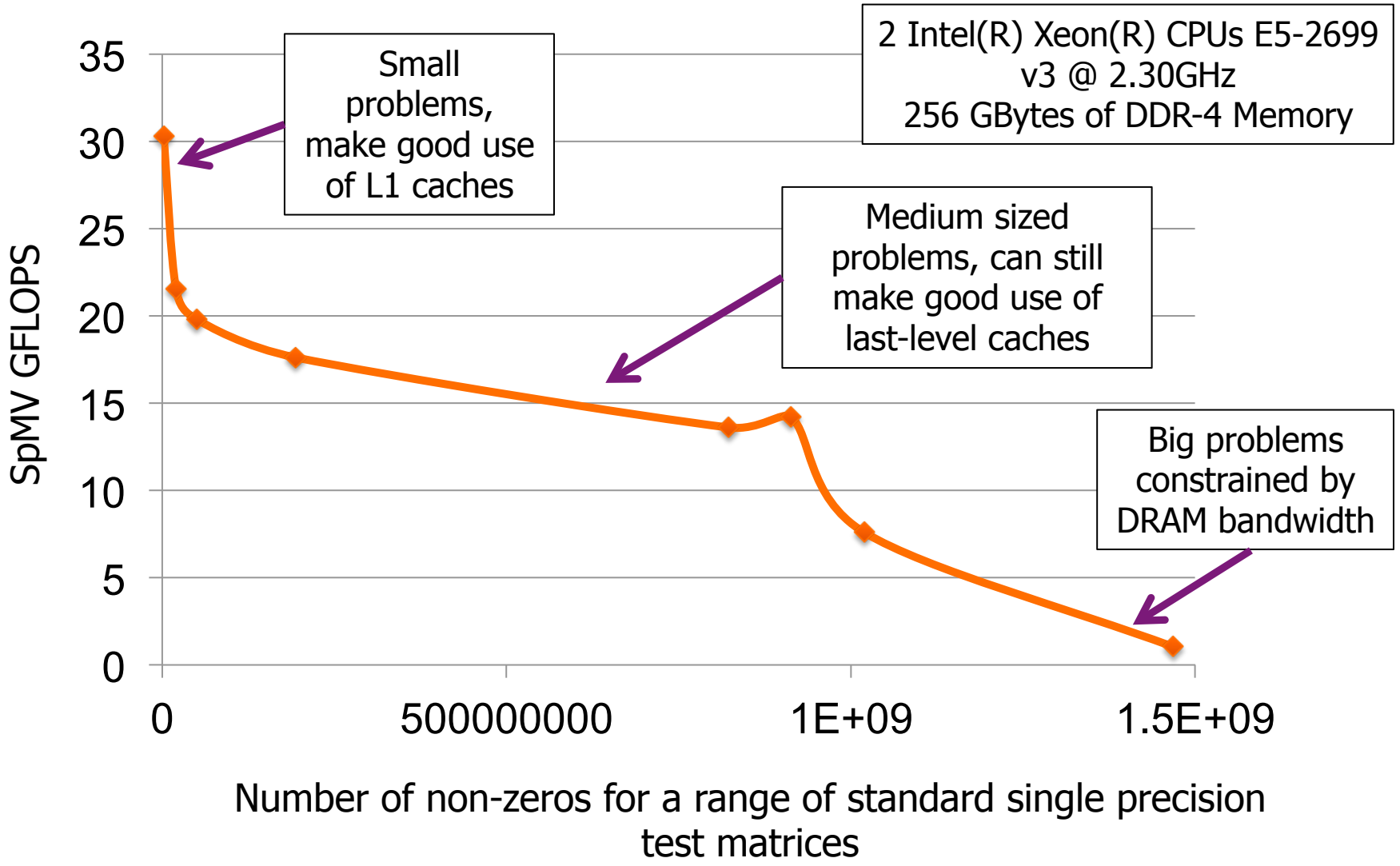
Memory Hierarchies

- A typical microprocessor memory hierarchy



  - Instruction cache and data cache pull data from a unified cache that maps onto DRAM.
  - TLB implements virtual memory and brings in pages to support large memory foot prints.

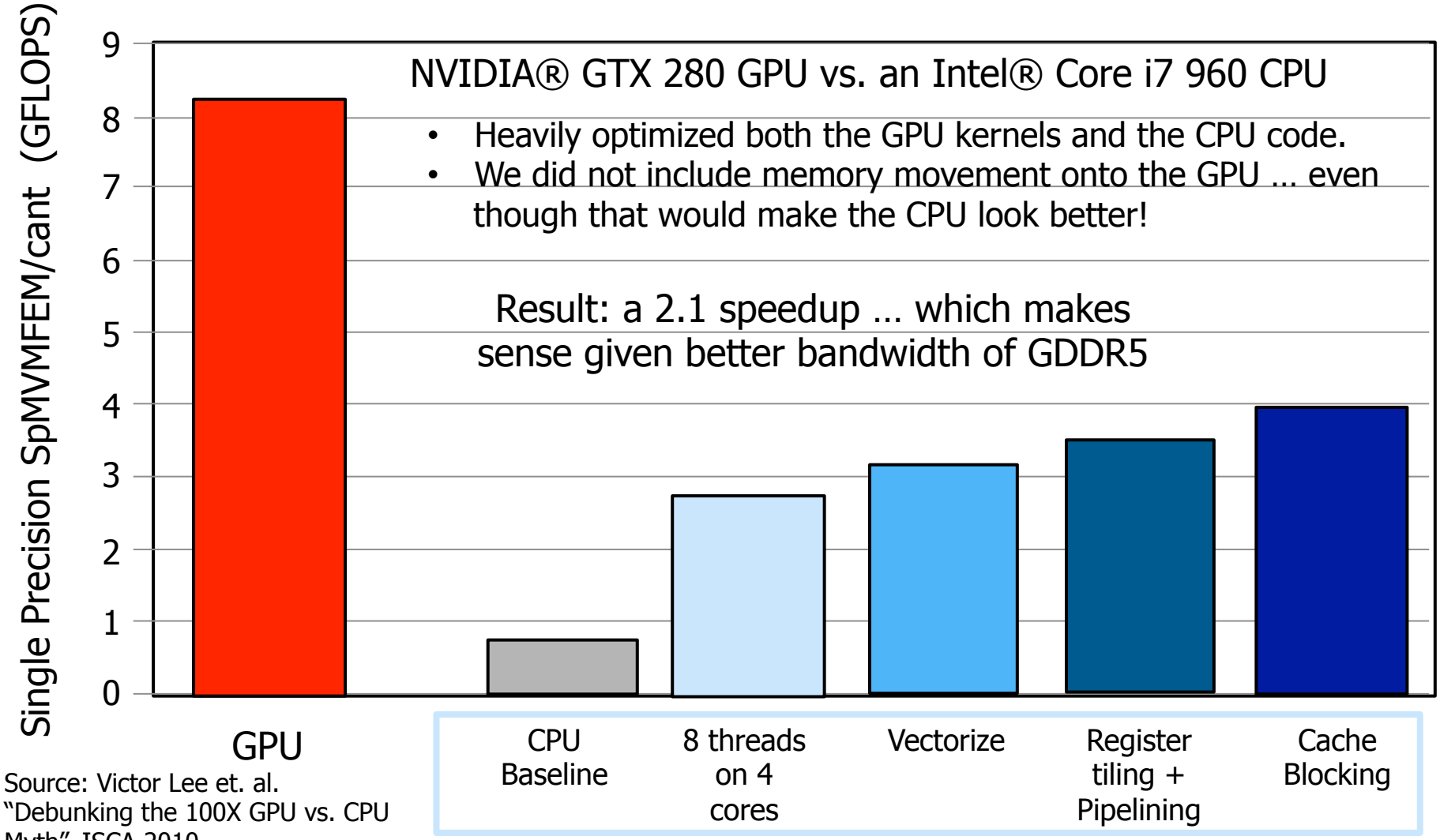
Sparse matrix vector multiplication



Source: Intel.
Intel® MKL™ 11.3, "mkl_cspblas_scsrgemv" with KMP_AFFINITY='verbose,granularity=fine,compact,1,0'

Sparse matrix vector product: GPU vs. CPU

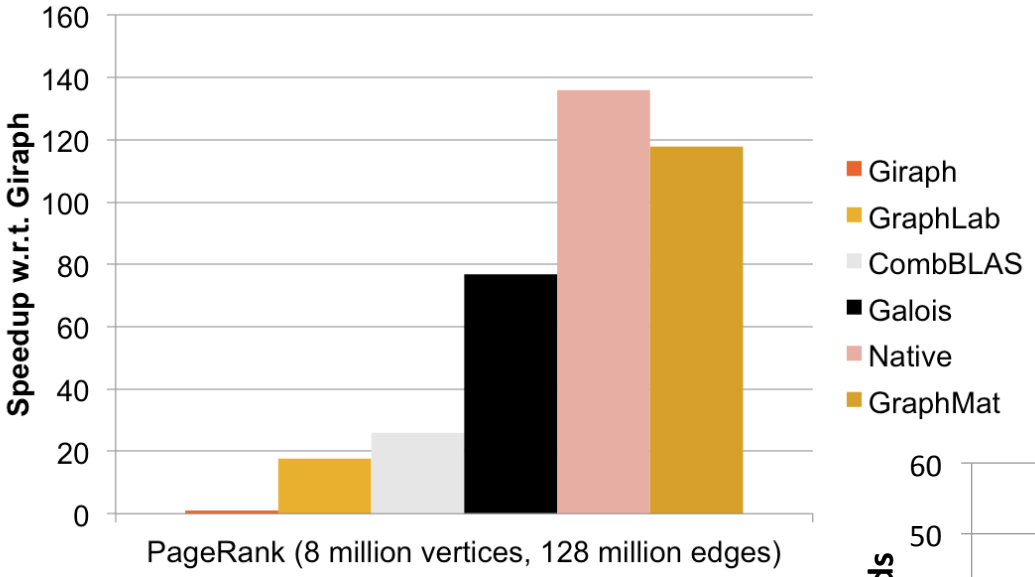
- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



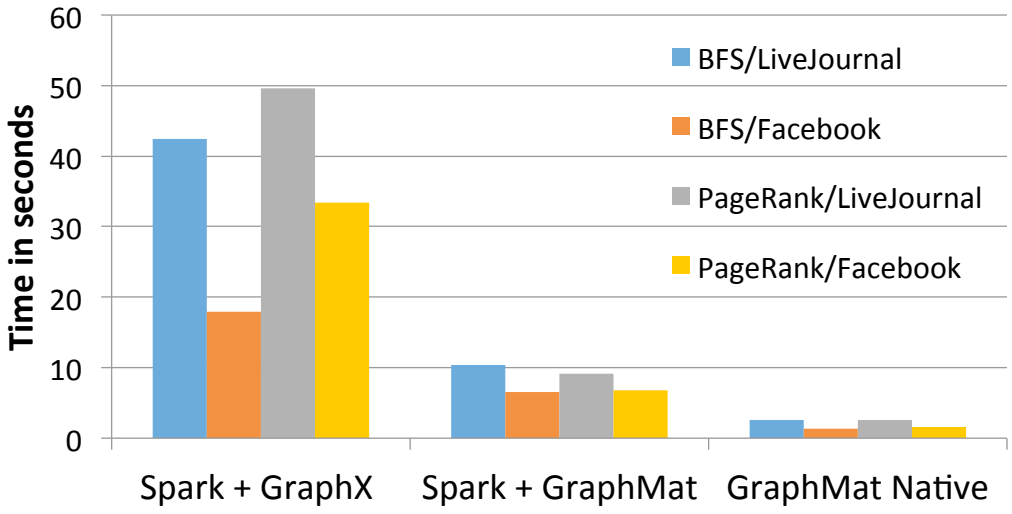
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Graph Analytics and Linear Algebra in action

GraphMAT is a project in Intel's Parallel Computing lab … The Power of sparse matrix algorithms with a programmer friendly "think like a vertex" front end.



Experiments run on a dual socket Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz



The term "native" refers to an implementation of the algorithm using low level, detailed hand-coded optimizations to the platform (i.e. high-level portable platforms were not used)

*BFS: Breadth First Search

PageRank 10 iterations, LiveJournal 85 million edges, Facebook 42 million edges. Data cached in 32-partition Spark RDD.

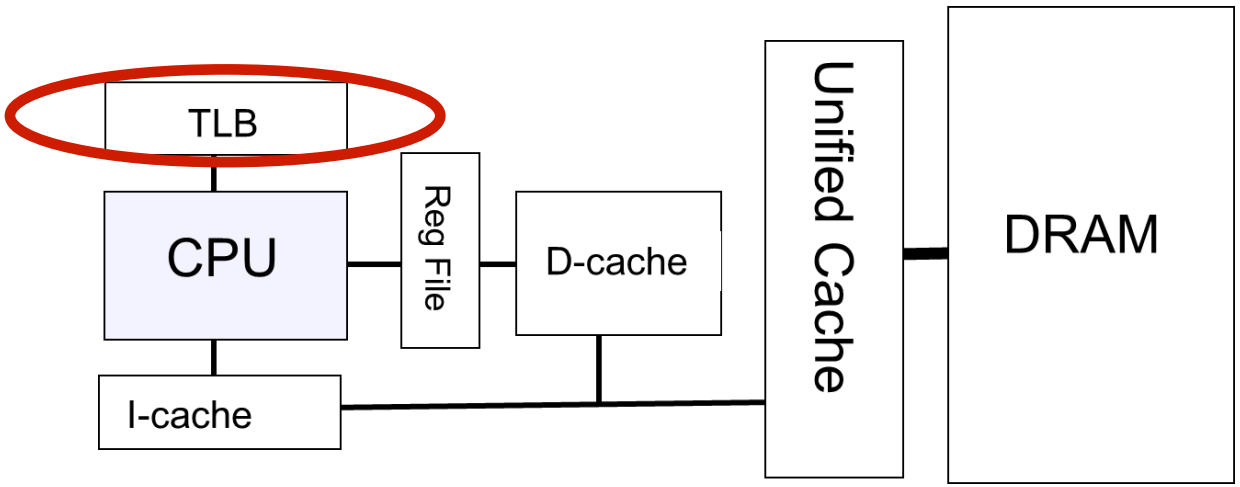
Consider the “simple” Matrix Transpose

- Copy the transpose of A into a second matrix B.

$A \in R^{NxN}$ $B \in R^{NxN}$

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

- Consider this operation and how it interacts with the TLB.



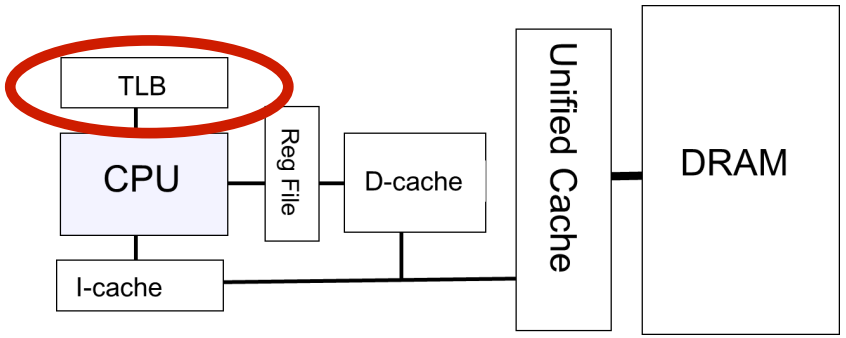
Consider the "simple" Matrix Transpose

- Copy the transpose of A into a second matrix B.

$A \in R^{NxN}$ $B \in R^{NxN}$

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

- TLB caches the mapping from virtual addresses to physical addresses
  - If the entry is in the page table, a small overhead of reading the entry and computed the physical address is incurred.
  - If the entry is not in the page table, a page fault exception will be raised. Requires expensive OS involvement and stalls the CPU.



Will the TLB impact performance of our transpose operation?

Consider the "simple" Matrix Transpose

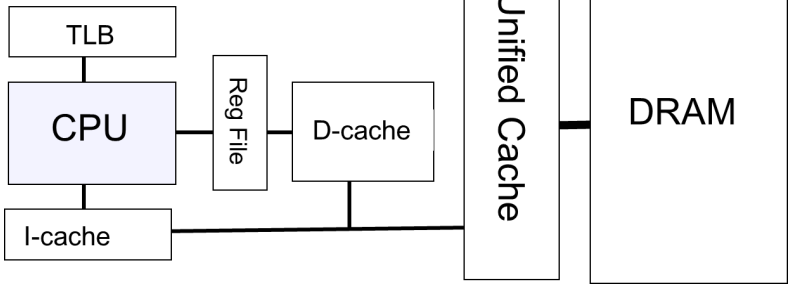
- Copy the transpose of A into a second matrix B.

$A \in R^{NxN}$ $B \in R^{NxN}$

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

For large N, as you march across addresses of A and B, you span multiple pages in memory … causes multiple page faults

- TLB caches the mapping from virtual addresses to physical addresses
  - If the entry is in the page table, the overhead of reading the entry and computed the physical address is needed
  - If the entry is not in the page table, a page fault exception will be raised. Requires expensive OS involvement and stalls the CPU.



Consider the “simple” Matrix Transpose

- Solution … break the loops into blocks so we reduce the number of page faults

Step 1: split the “i” and “j” loops into two … loops over tiles of a fixed size

```
for (i=0; i<N; i+=tile_size) {
      for (it=i; it<MIN(N,i+tile_size); it++){
 for (j=0; j<N; j+=tile_size) {
      for (jt=j; jt<MIN(N,j+tile_size);jt++){
            B[it+N*jt] = A[jt+N*it];
         }
       }
   }
}
```

Consider the "simple" Matrix Transpose

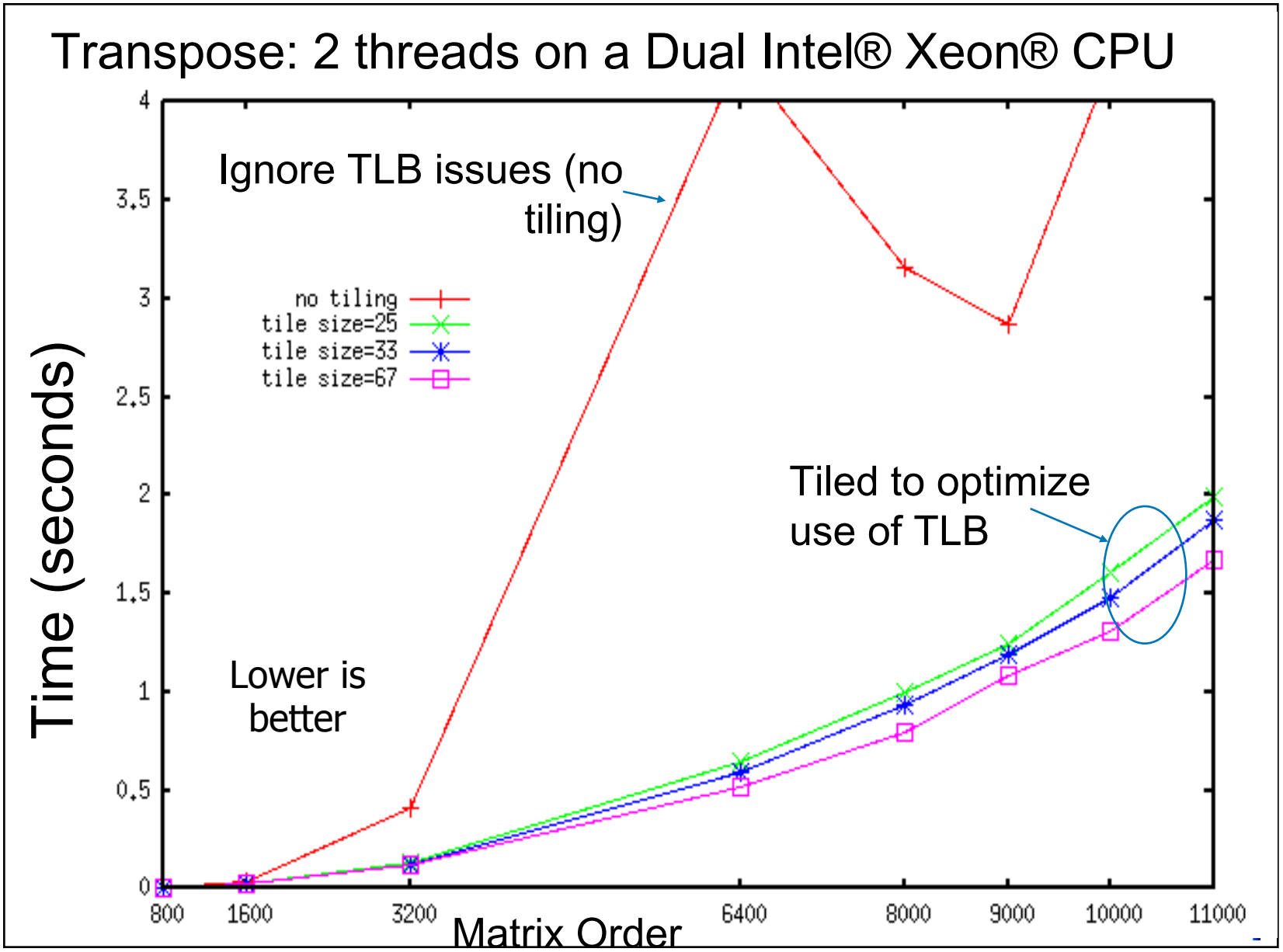
- Solution … break the loops into blocks so we reduce the number of page faults

Step 2: rearrange the loops moving the loops over the contents of tiles to the inner-most nest

```
for (i=0; i<N; i+=tile_size) {
for (j=0; j<N; j+=tile_size) {
    for (it=i; it<MIN(N,i+tile_size); it++){
        for (jt=j; jt<MIN(N,j+tile_size);jt++){
            B[it+N*jt] = A[jt+N*it];
          }
       }
   }
}
```

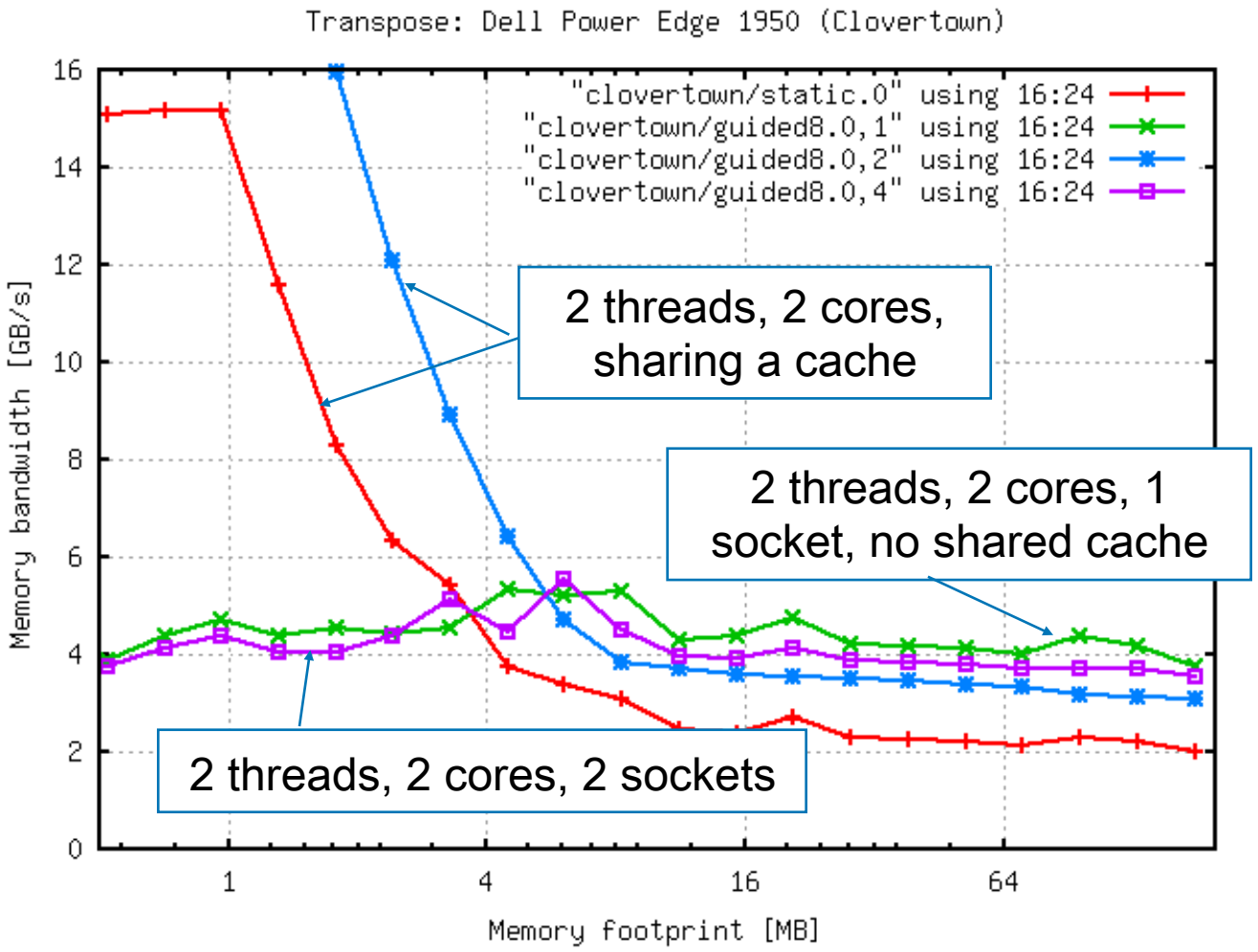
The result … you grab a tile, transpose that tile, then go to the next tile

Do you need to worry about the TLB?

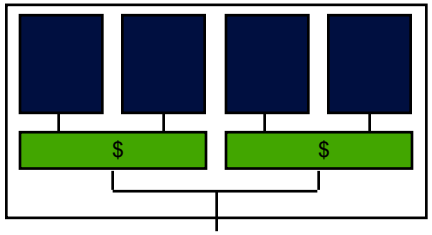


Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel

Thread Affinity and Transpose
2-socket Clovertown Dell PE1950



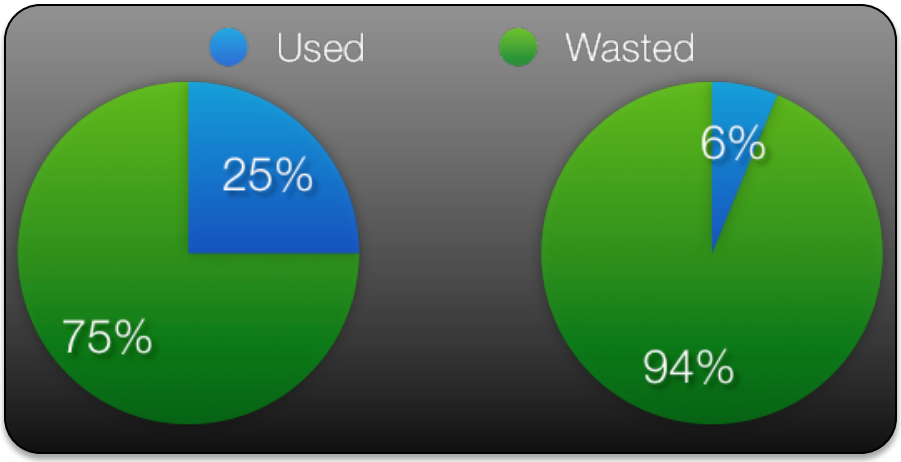
A single quad-core chip is a NUMA machine!



Xeon® 5300 Processor block diagram

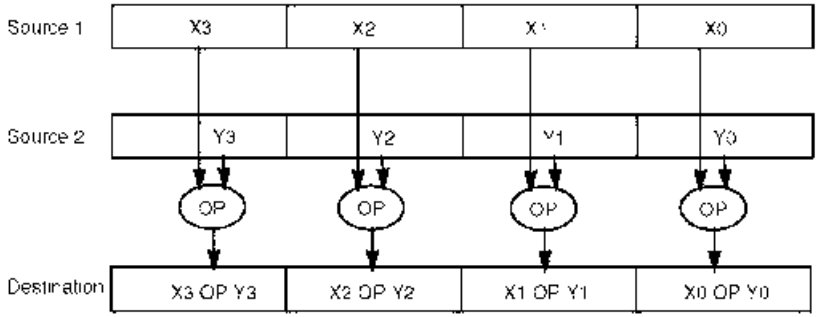
*NUMA == Non Uniform Memory architecture … memory is shared but access times vary.

Vector (SIMD) Programming



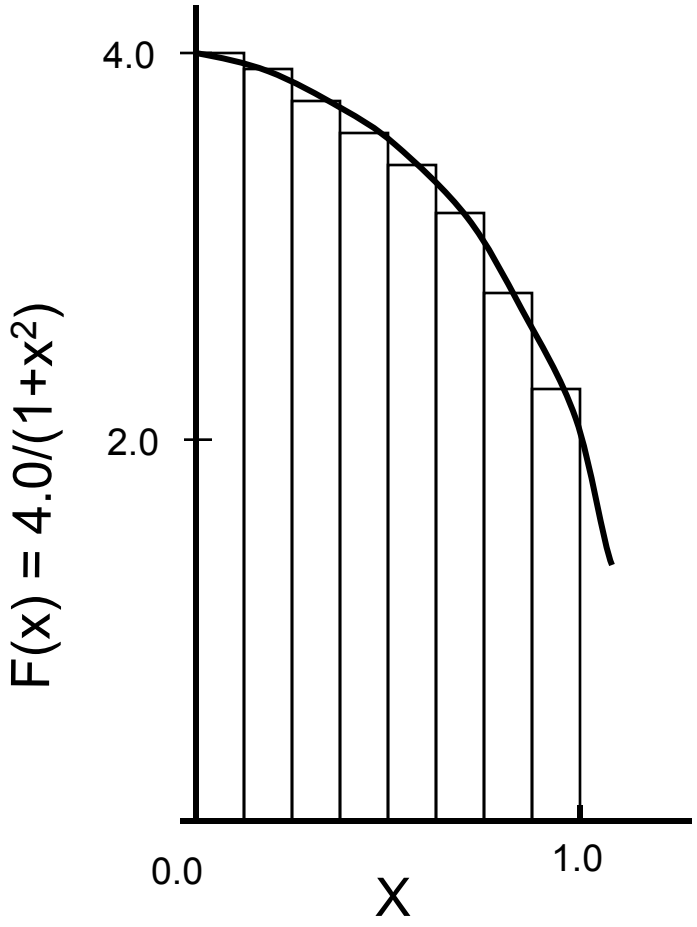
4 way SIMD (SSE)

16 way SIMD (Xeon™ PHI)



- Architects love vector units, since they permit space- and energy- efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit vectoriztion.
  - Use intrinsics … an assembly level approach.

Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Explicit Vectorization PI program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;
            #pragma omp simd reduction(+:sum)
            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Pi Program: Vectorization with intriniscs (SSE)

```
 float pi_sse(int  num_steps)
{
   int i,j;   float step, pi, vsum[4];
   float scalar_one   = 1.0, scalar_zero = 0.0;
   float ival, scalar_four  = 4.0;
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

 // unroll loop 4 times ... assume num_steps%4 = 0
   for (i=0;i< num_steps; i=i+4){
      ival  = (float)i;
      eye   = _mm_load1_ps(&ival);
      xvec  = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
      denom = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
      sum   = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);

   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

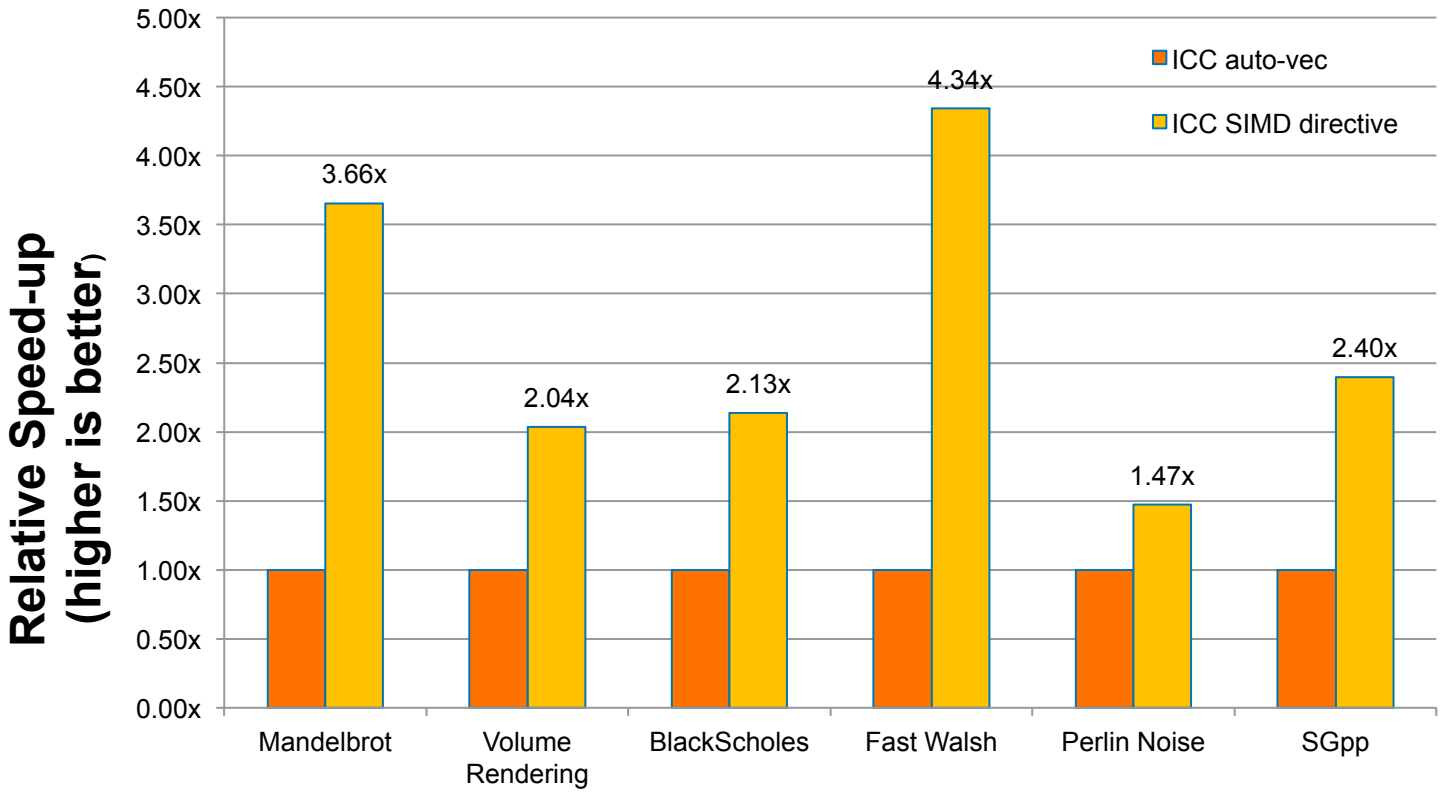
Method	Seconds for 8388608 steps
No vectorization (-O1)	0.0348
Autovectorization (-O3)	0.0151
Explicit vectorization	0.0169
Intrinsics	0.0042

Autovectorization (implicit) and explicit do about the same … half of peak (the SSE result).

But do you really want to write the SSE code?

Explicit Vectorization – Performance Impact

Explicit Vectorization looks better when you move to more complex problems.



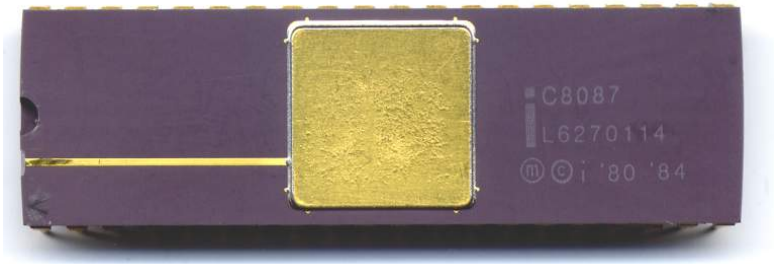
Source: M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP", pages 59-72, Rome, Italy, June 2012. LNCS 7312.

Outline

- The move to many core processors
- Why Linear Algebra is at the heart of data analytics
- Working with Many core CPUs
- Accelerators and the future of hardware
- A programmer's response to Hardware changes.

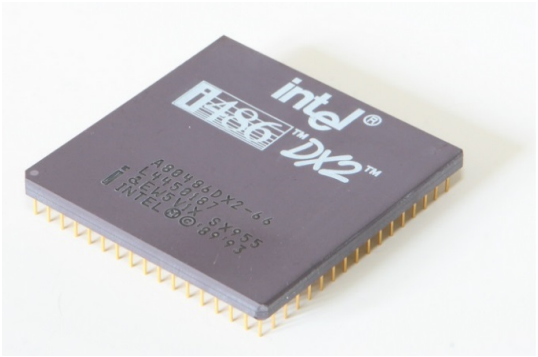
Coprocessors to accelerate flops

- The coprocessor: 8087 introduced in 1980.
  - The first x87 floating point coprocessor for the 8086 line of microprocessors.
  - Performance enhancements: 20% to 500%, depending on the workload.

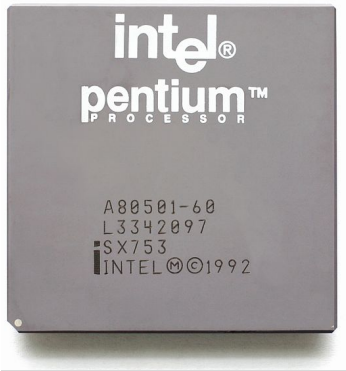


- Related Standards:
  - Partnership between industry and academia led to IEEE 754 …. The most important standard in the history of HPC. IEEE 754 first supported by x87.

- Intel® 80486DX, Pentium®, and later processors include floating-point functionality in the CPU … the end of the line for the X87 processors.



Intel® 486DX2™ processor, March 1992.

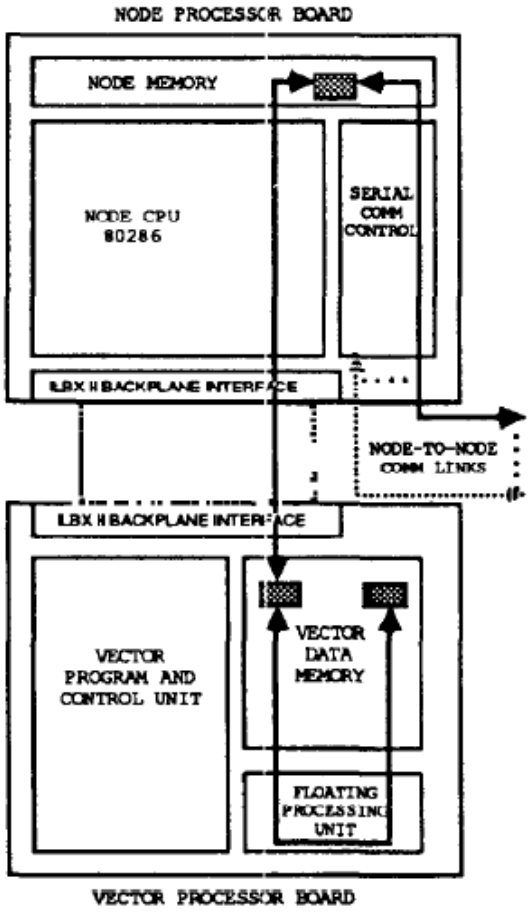


Intel® Pentium™ processor, Spring 1993.

Intel C8087 Math Coprocessor. Source: http://www.cpu-collection.de/?l0=i&i=1698&sd=1

Vector processing accelerators

- Coprocessor:
  - Vector co-processor (from Sky computer) for Intel iPSC/2 MPP (~1987)
- Related Standards
  - Compiler vectorization… not a standard but a universally expected capability.
  - OpenMP 4.0 recently added pragmas for explicit vectorization.
- The Intel® i860 processor introduced in the early 90's and used in Intel's Paragon Supercomputers.
  - 25 MHz to 50 MHz
  - Included a SIMD unit to accelerate graphics operations
  - Had a strong influence on later MMX vector instructions



Source: Implementation of the conjugate gradient algorithm on a vecor hypercube multiprocessor, Aykanat, Ozguner, Scott

Source: http://www.cpu-collection.de/?l0=co&l1=Intel&l2=i860

*third party names are the property of their owners

Coprocessors to support Graphics (and more)

1st generation:
Voodoo 3dfx (1996)

2nd Generation:
GeForce 256/Radeon 7500
(1998)

3rd Generation: GeForce3/Radeon 8500 (2001). The first GPU to allow a limited programmability in the vertex pipeline.

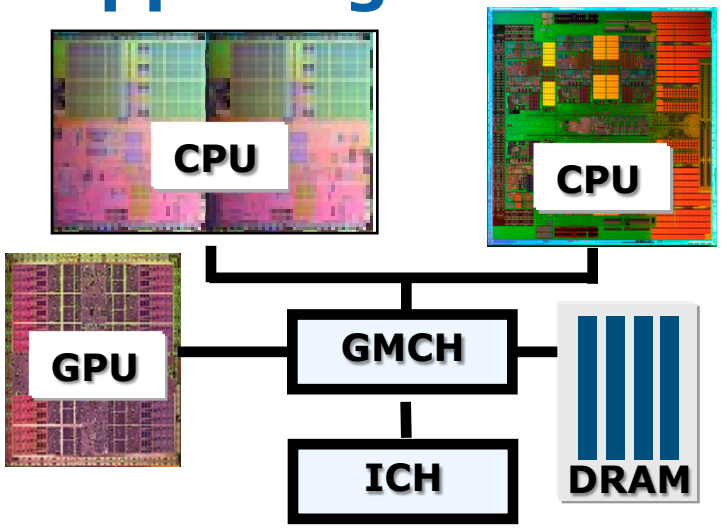
4th Generation: Radeon 9700/GeForce FX (2002): The first generation of "fully-programmable" graphics cards.

5th Generation: GeForce 8800/HD2900 (2006) and the birth of CUDA

Standards?

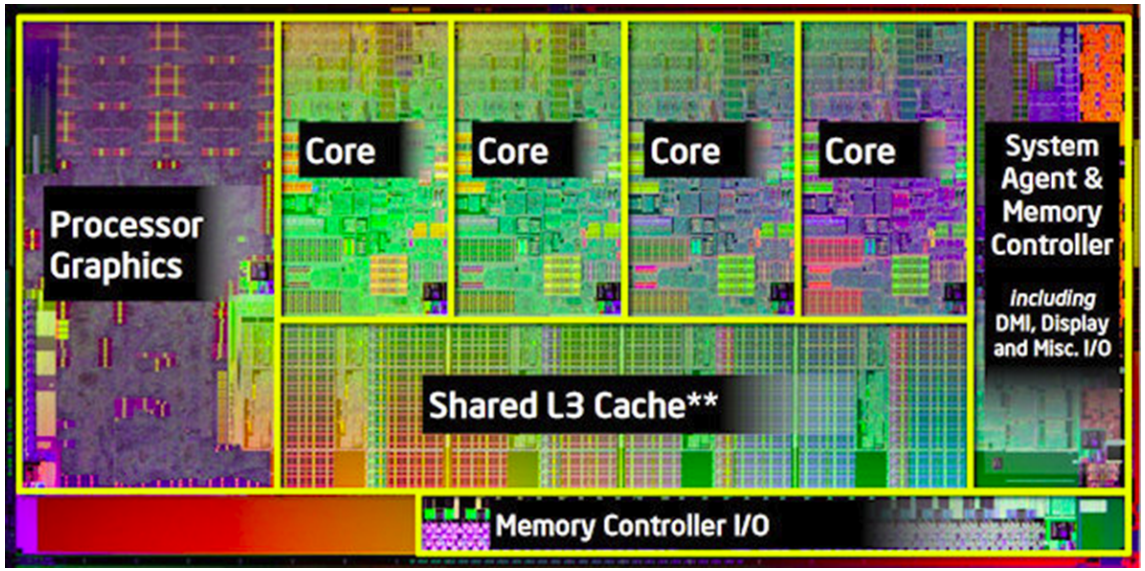
- OpenGL for graphics … quite successful
- OpenCL for GPGPU … not as successful as I'd like
- OpenACC is not a vendor-neurtal standard … OpenMP 4.0 is he right path for directive driven GPGPU programming.

The absorption of the GPU into the CPU is happening now



- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

- Current designs put this functionality onto a single chip … mitigates the PCIe bottleneck in GPGPU computing!



Intel® Core™ i5-2500K Desktop Processor (Sandy Bridge) Intel HD Graphics 3000 *(2011)*

GMCH = graphics memory control hub,
ICH = Input/output control hub

Blurring the line between the GPU and the CPU

Knights Landing: Next Intel® Xeon Phi™ Processor

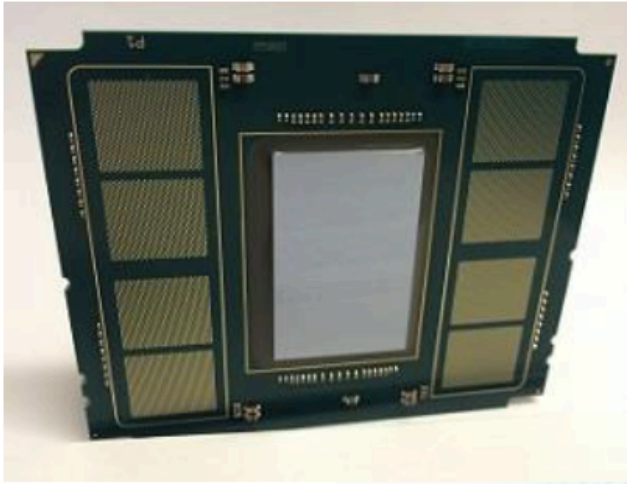
Intel® Many-Core Processor targeted for HPC and Supercomputing

First **self-boot** Intel® Xeon Phi™ processor that is **binary compatible** with main line IA. Boots standard OS.

Significant improvement in scalar and **vector** performance

Integration of **Memory on package**: innovative memory architecture for high bandwidth and high capacity

Integration of **Fabric on package**



Three products

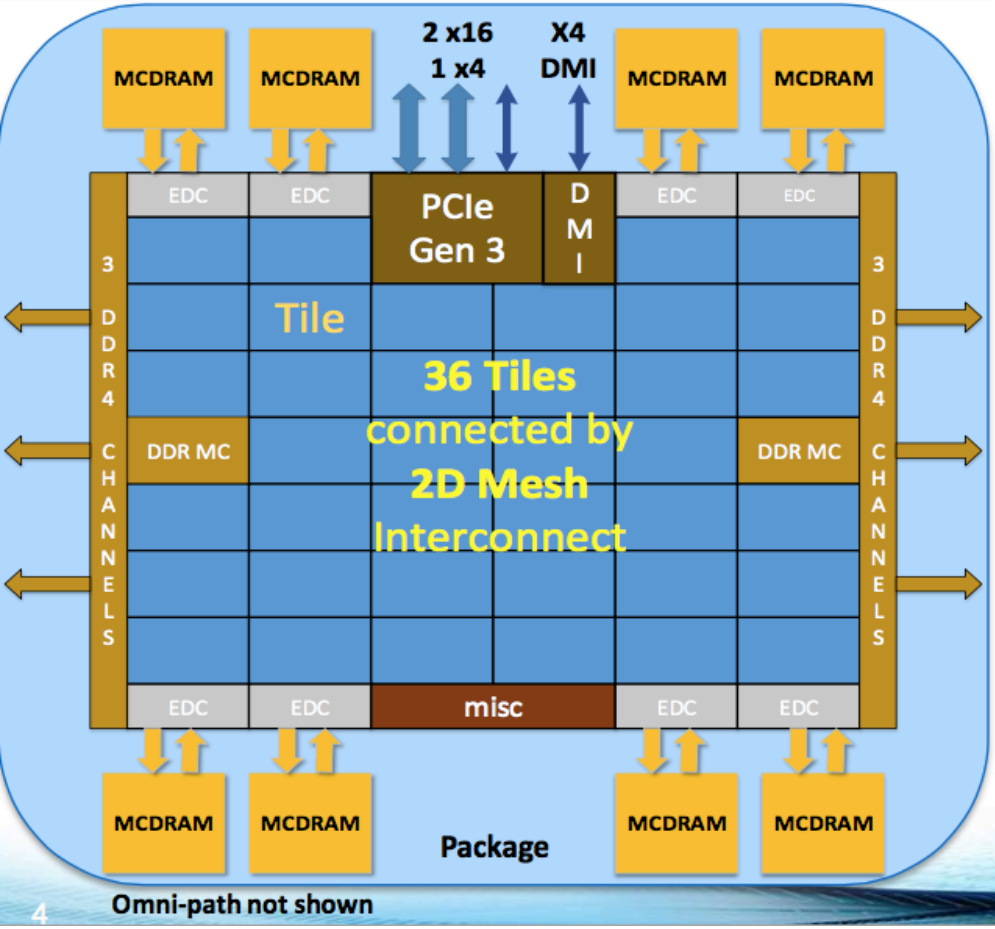
KNL Self-Boot	KNL Self-Boot w/ Fabric	KNL Card
(Baseline)	(Fabric Integrated)	(PCIe-Card)

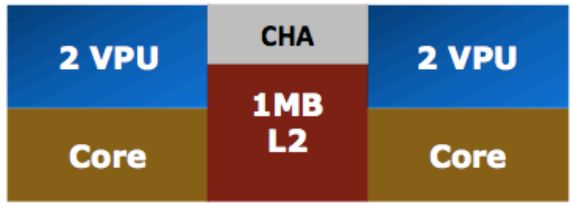
Potential future options subject to change without notice.
All timeframes, features, products and dates are preliminary forecasts and subject to change without further notification.

Source: Avinash Sodani, Hot Chips 2015 KNL talk

Lots of cores, each with a pair of SIMD units

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

4

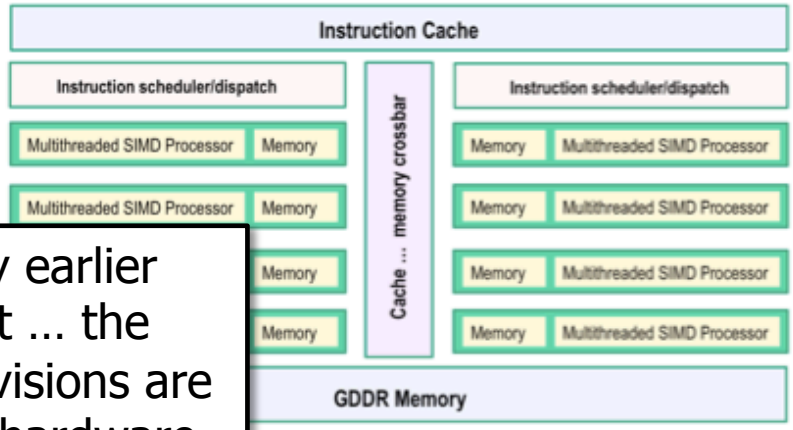
Source: Avinash Sodani, Hot Chips 2015 KNL talk

Outline

- The move to many core processors
- Why Linear Algebra is at the heart of data analytics
- Working with Many core CPUs
- Accelerators and the future of hardware
- A programmer's response to Hardware changes.

The software platform debate!

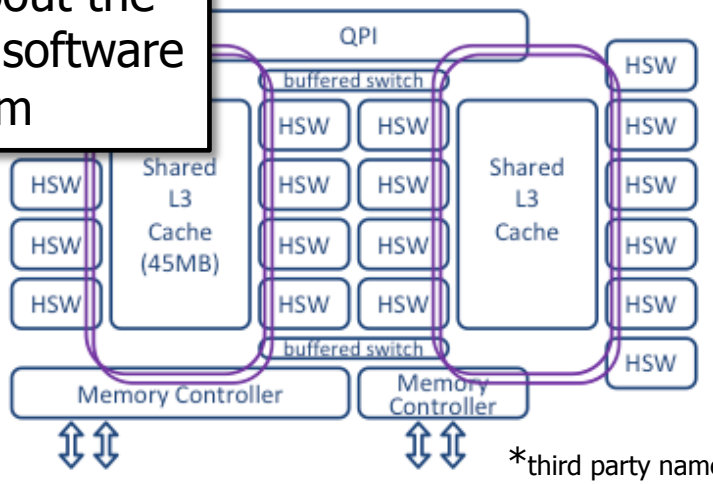
It's really about competing software platforms



GPU

- **Single Instruction multiple threads.**
  - turn loop bodies into kernels.
  - HW intelligently schedules kernels to hide latencies.
- *Dogma*: a natural way to express huge amounts of data parallelism
- Examples: CUDA, OpenCL, OpenACC

Recall my earlier comment … the competing visions are less about hardware and more about the programmers software platform

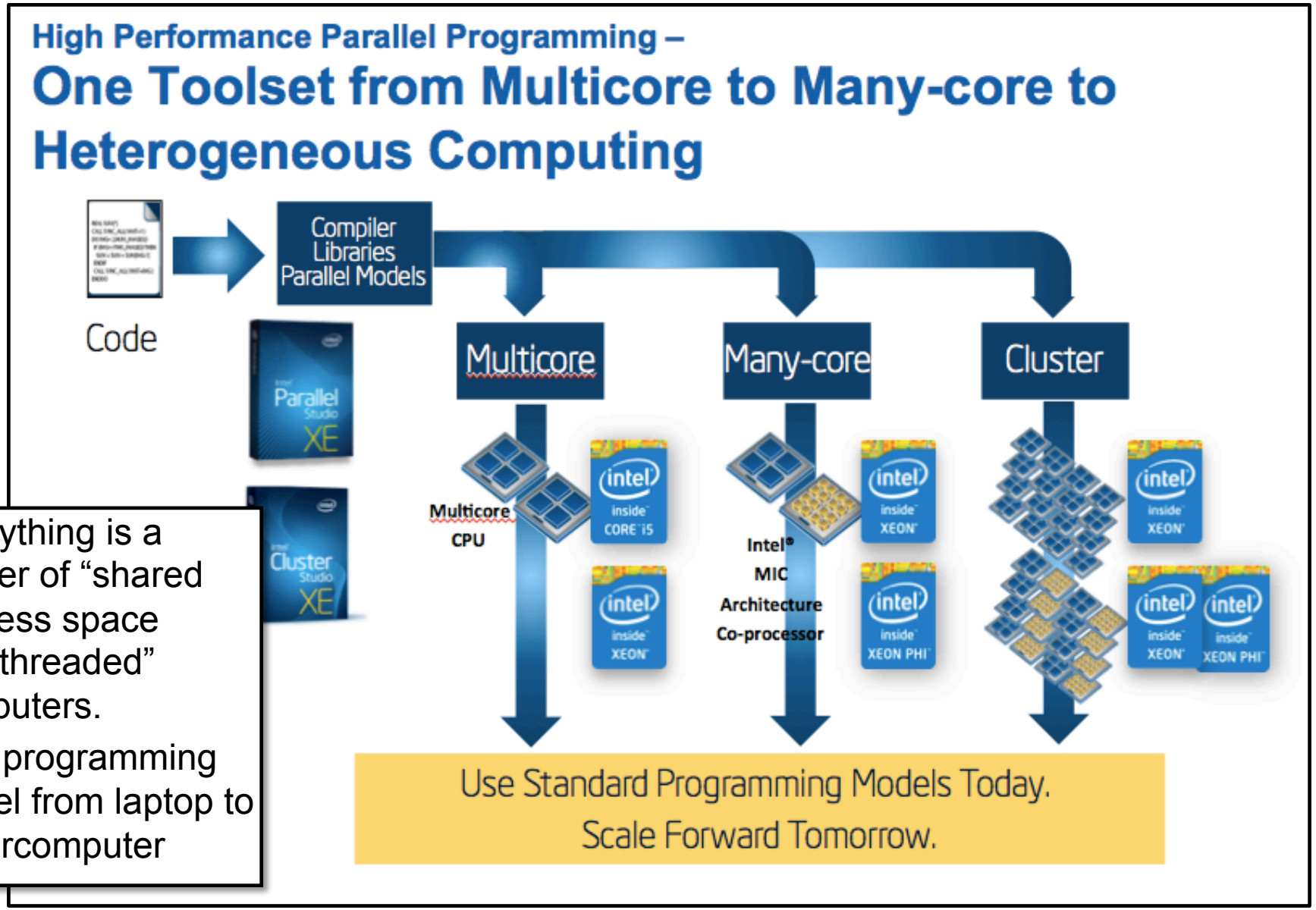


CPU

- **Shared Address space, multi-threading.**
  - Many threads executing with coherent shared memory.
- *Dogma*: The legacy programming model people already know. Easier than alternatives.
- Examples: OpenMP, Pthreads, C++11

*third party names are the property of their owners

The official strategy at Intel.



- Everything is a cluster of "shared address space multithreaded" computers.
- One programming model from laptop to supercomputer

What about the competing platform:
Single Instruction multiple thread (SIMT)?

- Dominant as a proprietary solution based on CUDA and OpenACC.
- But there is an Open Standard response (supported to varying degrees by all major vendors)

OpenCL

SIMT programming for CPUs, GPUs, DSPs, and FPGAs. Basically, an Open Standard that generalizes the SIMT platform pioneered by our friends at NVIDIA®

OpenMP™

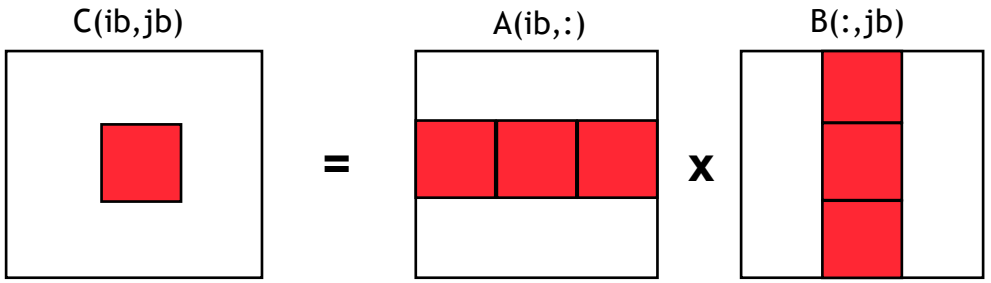
OpenMP 4.0 added target and device directives … Based on the same work that was used to create OpenACC. Therefore, just like OpenACC, you can program a GPU with OpenMP!!!

The long term viability of the SIMT platform depends on the user community demanding (and using) the Open Standard alternatives!

*third party names are the property of their owners

Portable performance: dense matrix multiplication

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```



Transform the basic serial matrix multiply into multiplication over blocks

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
   Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

Load A and B blocks, wait for all work-items to finish

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
     Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     Abase += Ainc;     Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance (in MFLOPS)

- Single Precision matrix multiplication (order 1000 matrices)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Same identical code running on multiple CPUs and GPUs without change (just recompile). This is THE BEST performance portability I've ever seen!!!!

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB
* The comp was run twice and only the second time is reported (hides cost of memory movement.

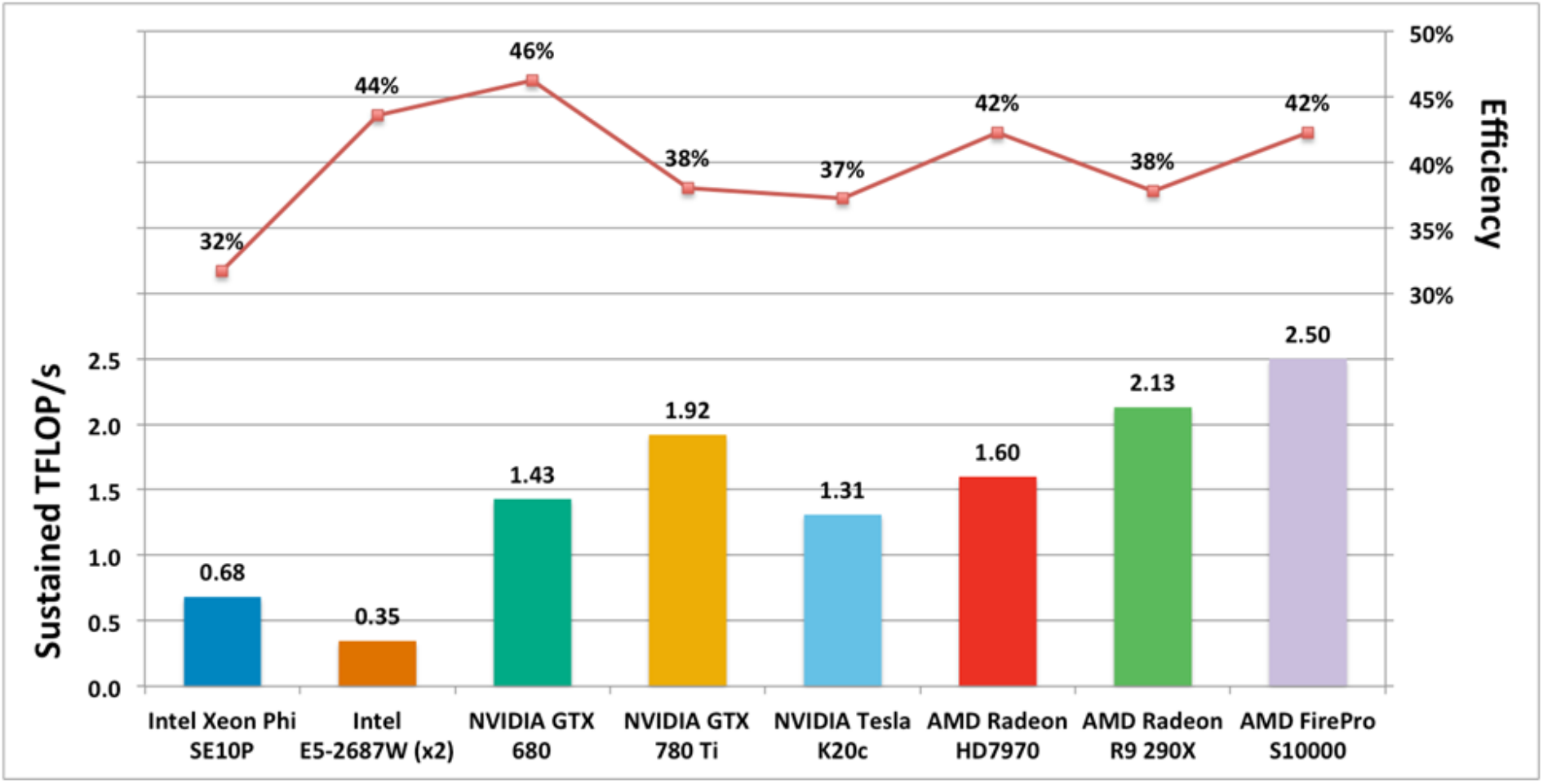
Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

BUDE: Bristol University Docking Engine

One program running well on a wide range of platforms



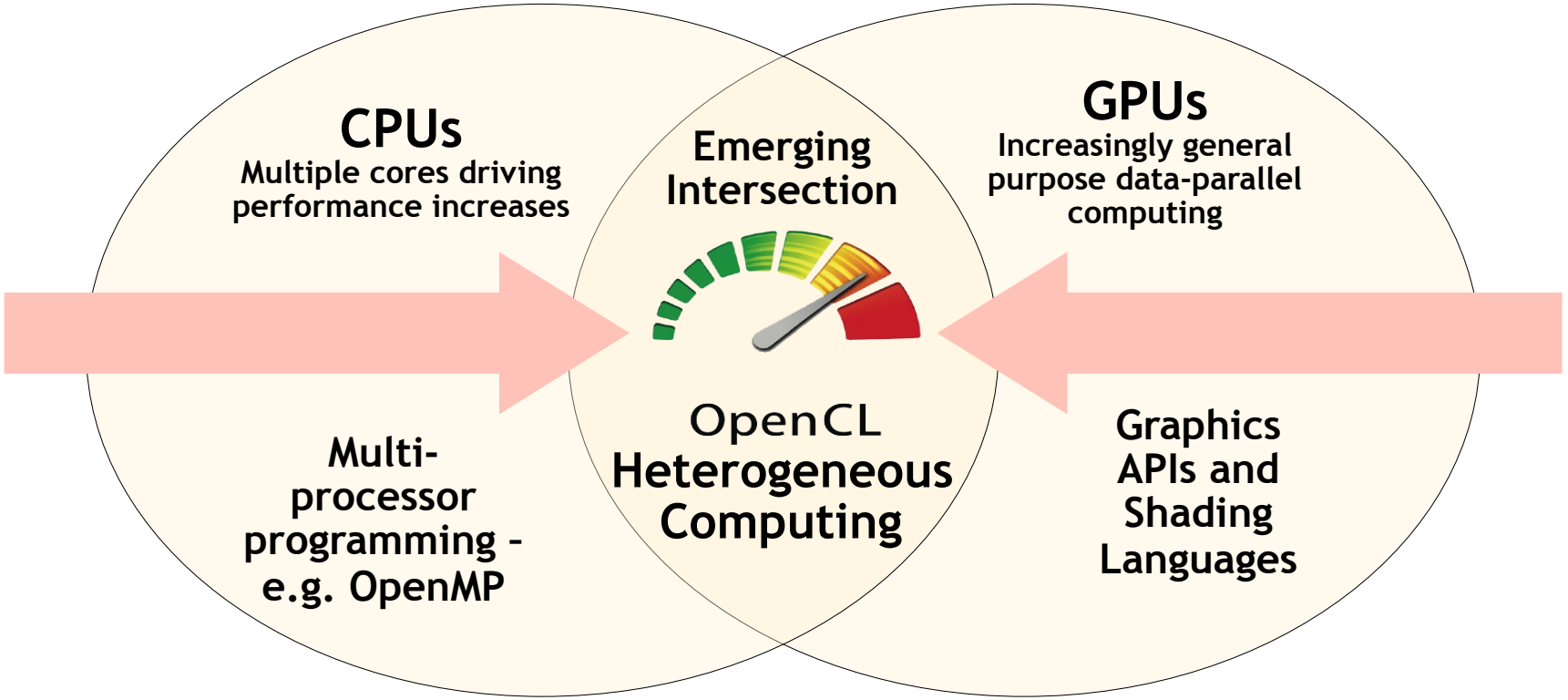
Summary

- Physics is driving the move to many core chips so it is here to stay. Adapt or get left behind.
- For a CPU, the key issues are:
  - Program to the memory hierarchy
  - Thread affinity
  - vectorization
- The CPU vs GPU debates are not really about hardware … those differences are shrinking as the GPU gets sucked into the CPU.
- The key is the software platform … shared address space multi-threading vs. Single Instruction Multiple Threads (SIMT).
- The jury is out as to whether the standards required to support SIMT in the long run will be adopted … its in the hands of the application programming community, not the vendors.

Back up

- Details about the OpenMP matrix multiplication example.
- Debunking the GPU/CPU myths … a few more results from Victor Lee's talk at ISCA 2010
- Additional information about Knights landing from the Hot chips talk in 2015

Industry Standards for Programming Heterogeneous Platforms

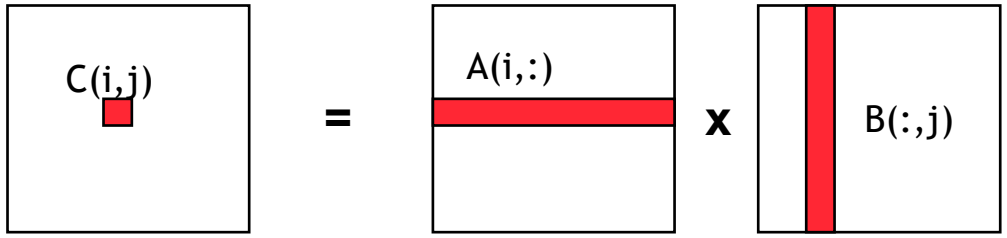


OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

Matrix multiplication example:
Naïve solution, one dot product per element of C

- Multiplication of two dense matrices.



Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks "out" and leave loops over a single block together

Matrix multiplication: sequential code

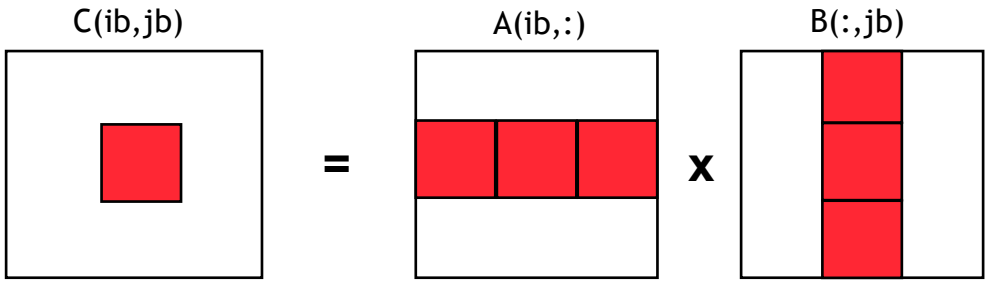
```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
```



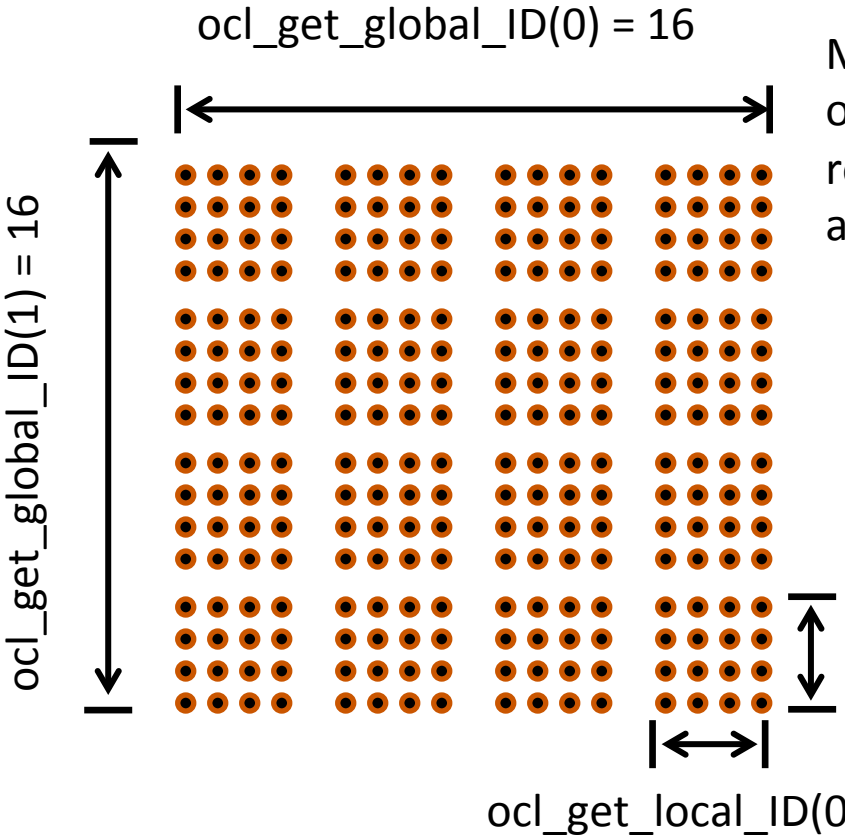

```
}
```

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Mapping into A, B, and C from each work item

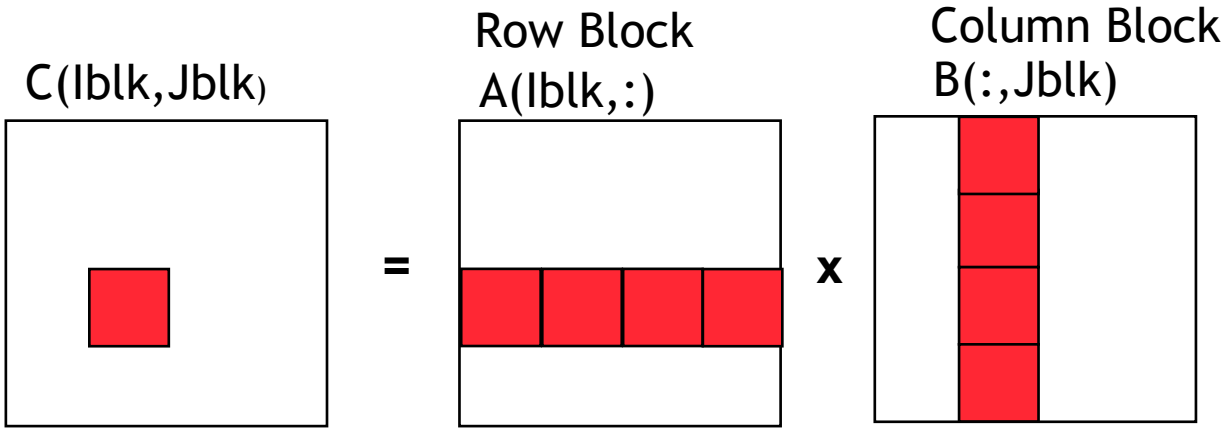
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



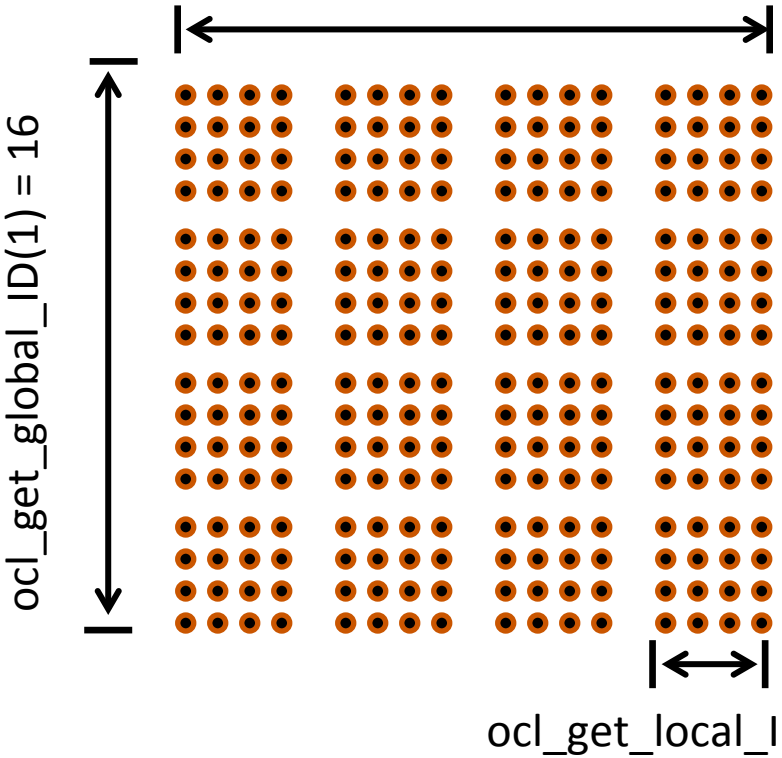
Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item



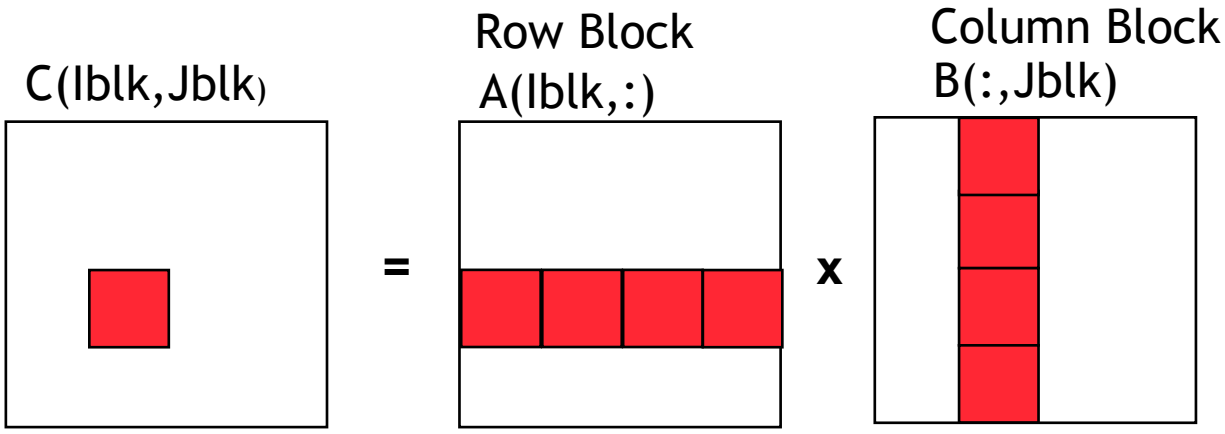
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item



Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4

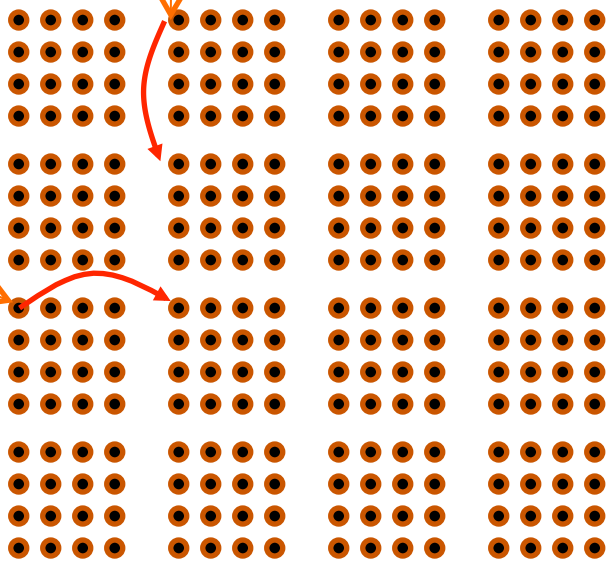
Consider indices for computation of the block C(Iblk=2, Jblk=1)

Bbase = Jblk*blksz = 1*4

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Abase = Iblk*N*blksz
= 1 * 16 * 4

Subsequent A blocks by shifting index by Ainc = blksz = 4



Subsequent B blocks by shifting index by Binc = blksz * N
= 4 * 16 = 64

Back up

- Details about the OpenMP matrix multiplication example.
- Debunking the GPU/CPU myths … a few more results from Victor Lee's talk at ISCA 2010
- Additional information about Knights landing from the Hot chips talk in 2015

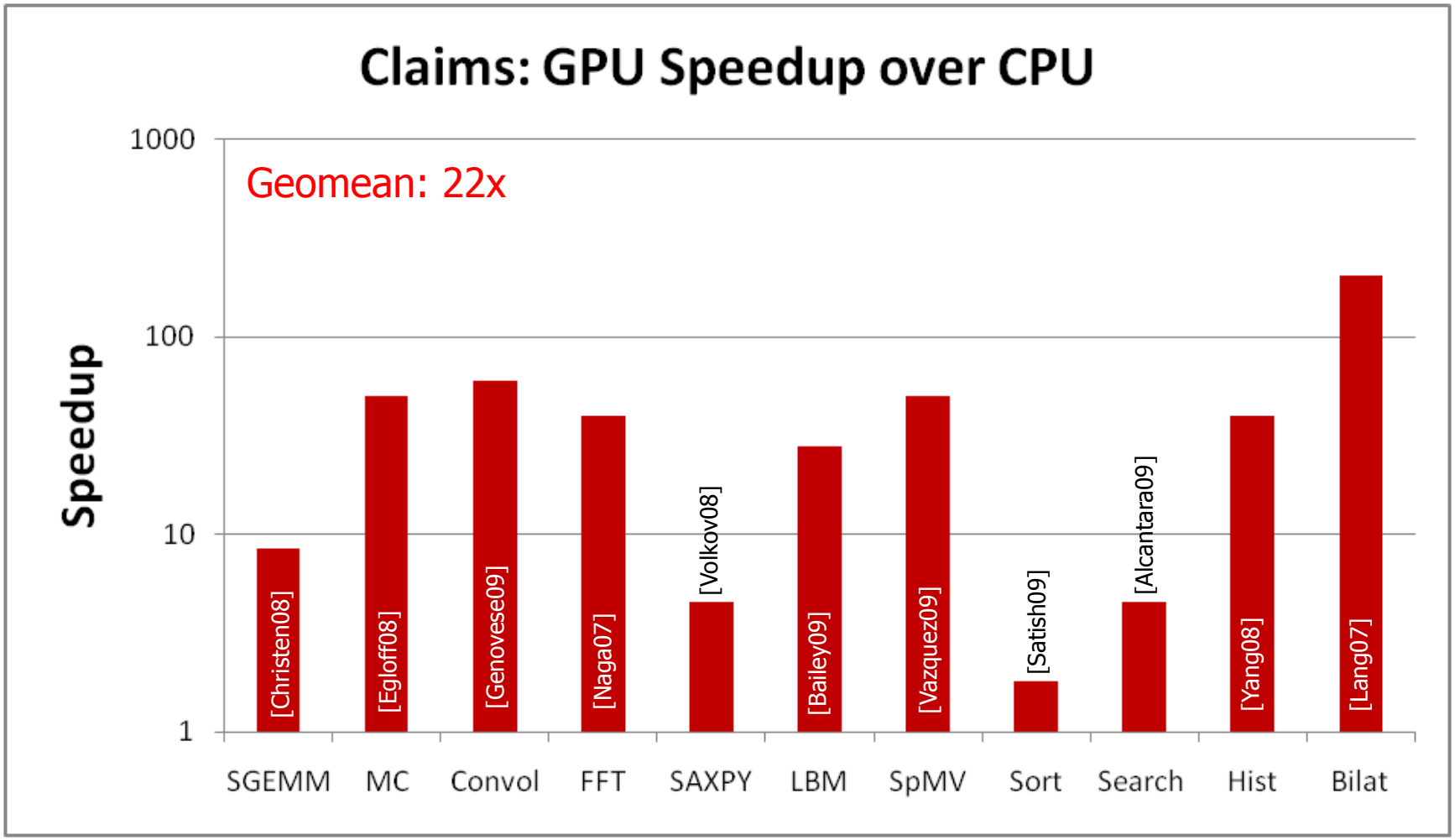
Methodology

- Start with previously best published code / algorithm
- Validate claims by others
- Optimize BOTH CPU and GPU versions
- Collect and analysis performance data

Note: Only computation time on the CPU and GPU is measured. PCIe transfer time and host application time are not measured for GPU. Including such overhead will lower GPU performance

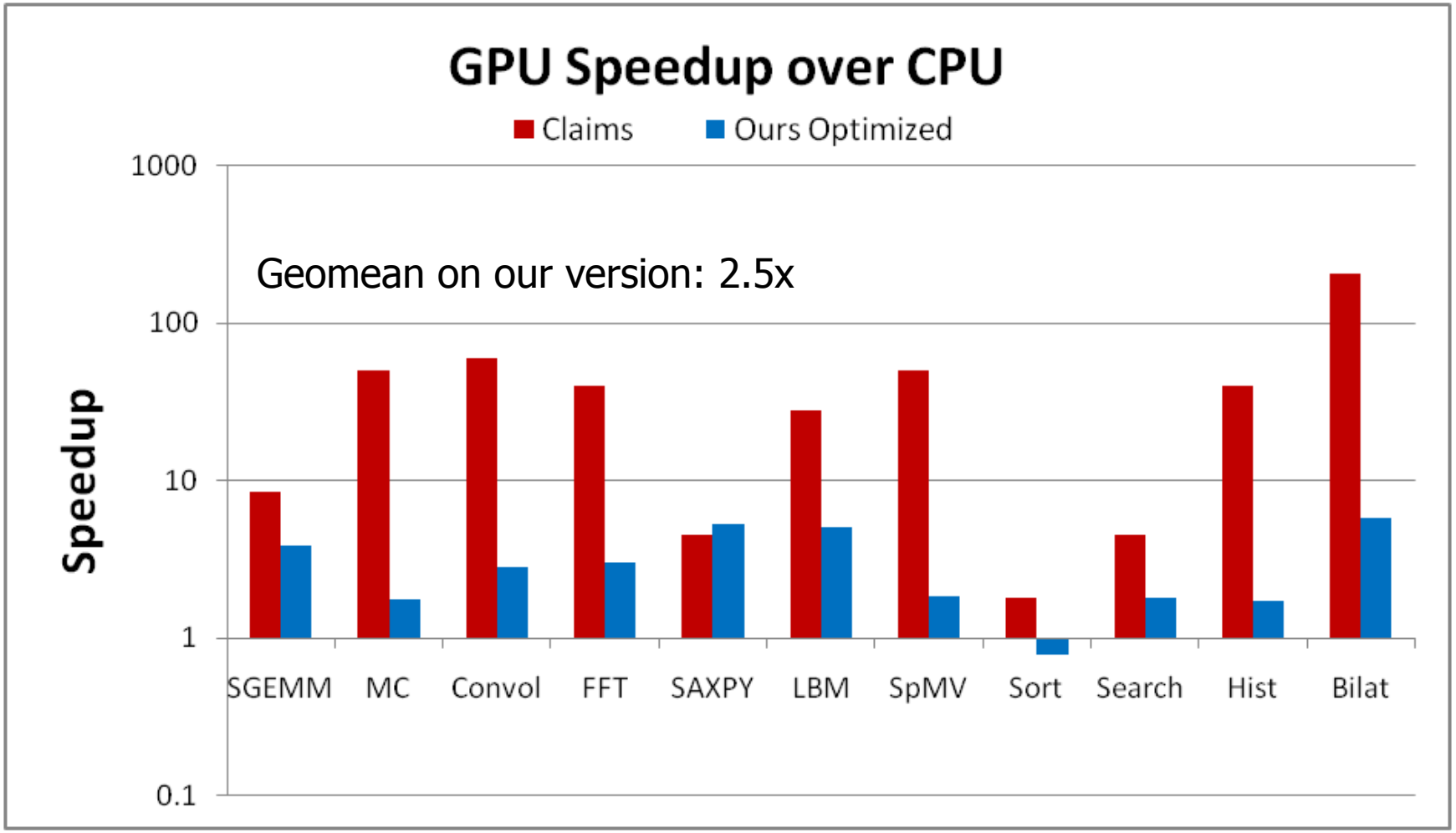
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What was claimed



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What we measured



Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

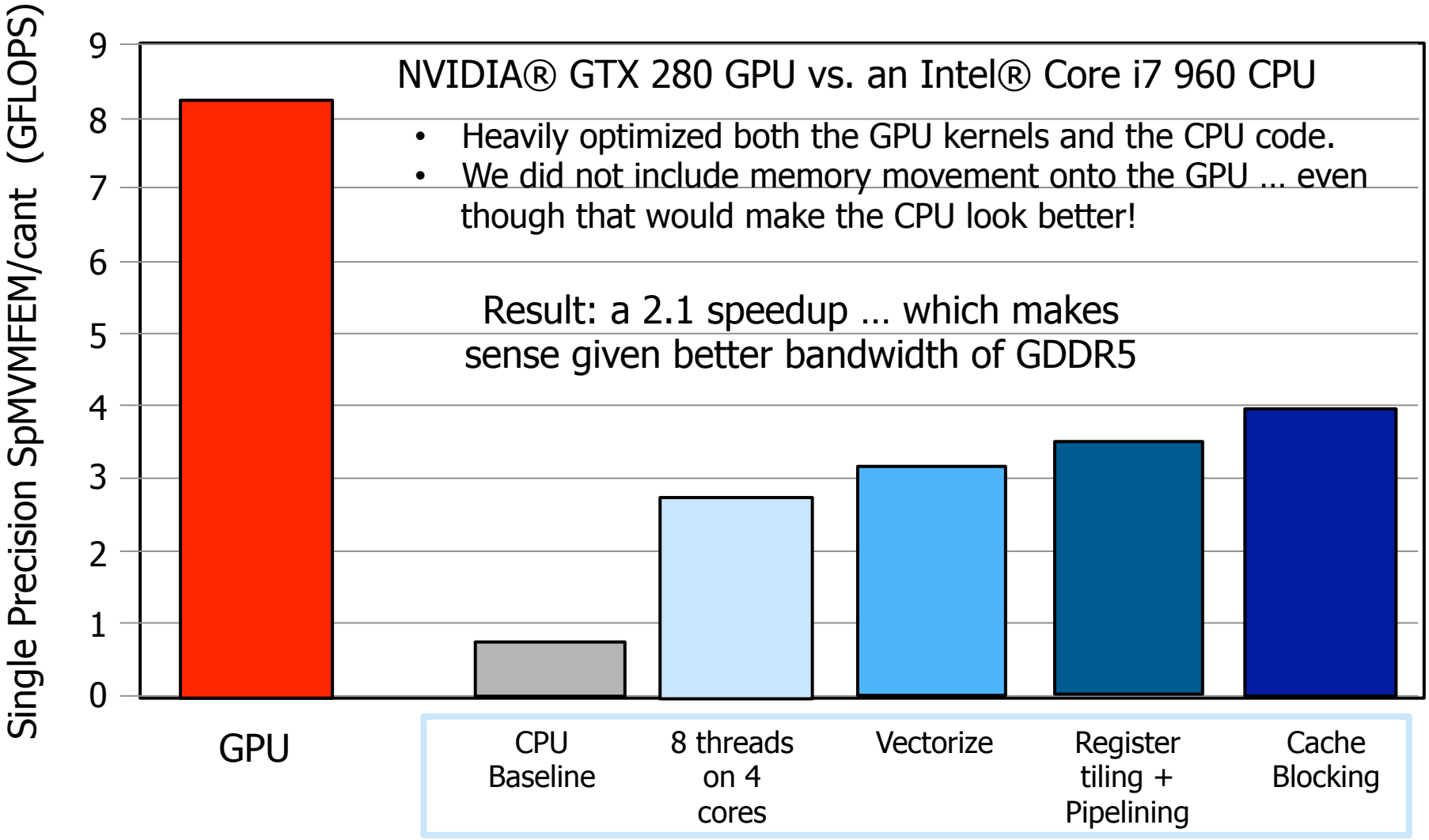
What went wrong

- CPU and GPU are not contemporary
- All attention is given to GPU coding
- CPU version is under optimized
  - E.g. Not using multi-threading
  - E.g. Not using common optimizations such as cache blocking

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Sparse matrix vector product

- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Back up

- Details about the OpenMP matrix multiplication example.
- Debunking the GPU/CPU myths … a few more results from Victor Lee's talk at ISCA 2010
- Additional information about Knights landing from the Hot chips talk in 2015

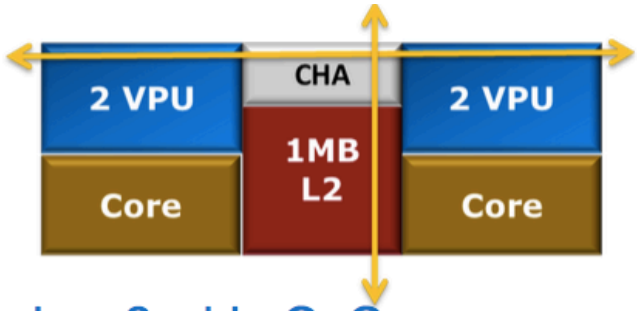
Please … read the following notice if you are going to look at the next few slides

Legal

Knights landing: official details part 1

KNL Tile: 2 Cores, each with 2 VPU

1M L2 shared between two Cores



Core: Changed from Knights Corner (KNC) to KNL. Based on 2-wide OoO Silvermont™ Microarchitecture, but with *many* changes for HPC.

4 thread/core. Deeper OoO. Better RAS. Higher bandwidth. Larger TLBs.

2 VPU: 2x AVX512 units. 32SP/16DP per unit. X87, SSE, AVX1, AVX2 and EMU

L2: 1MB 16-way. 1 Line Read and ½ Line Write per cycle. Coherent across all Tiles

CHA: **C**aching/**H**ome **A**gent. Distributed Tag Directory to keep L2s coherent. MESIF protocol. 2D-Mesh connections for Tile

Source: Avinash Sodani, Hot Chips 2015 KNL talk

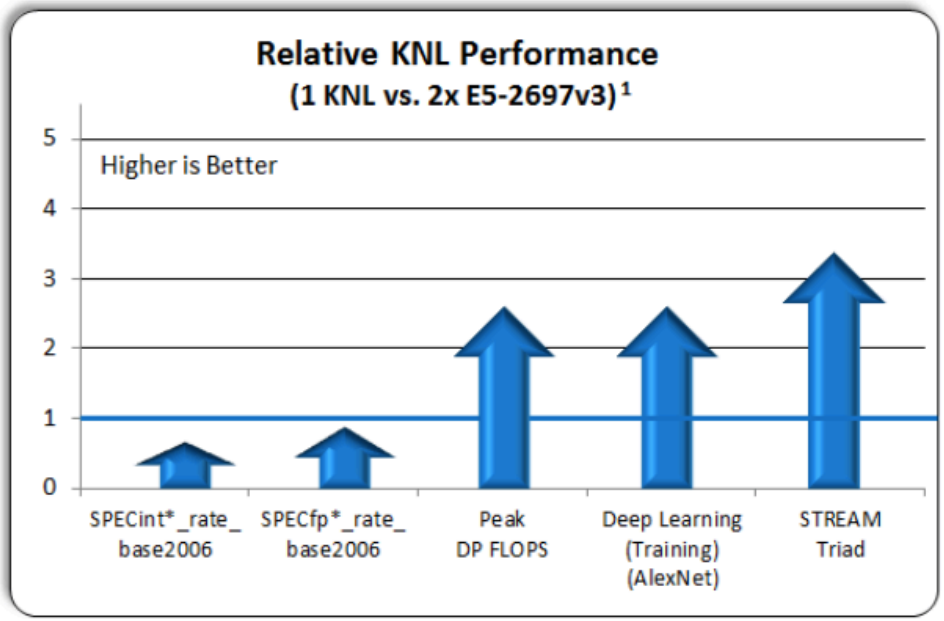
Knights Landing: official details part 2

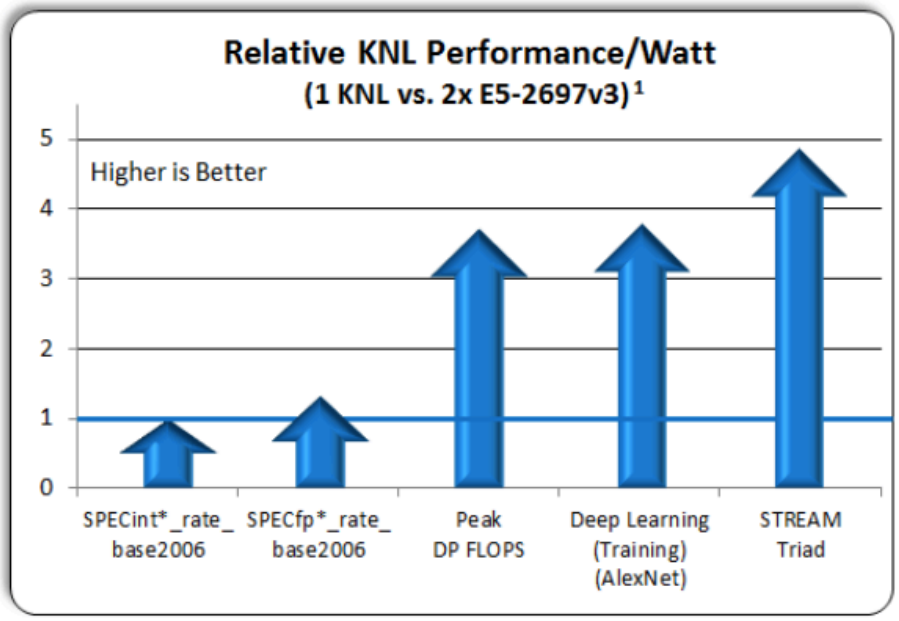
Many Trailblazing Improvements in KNL

Improvements	What/Why
Self Boot Processor	No PCIe bottleneck
Binary Compatibility with Xeon	Runs all legacy software. No recompilation.
New Core: Atom™ based	~3x higher ST performance over KNC
Improved Vector density	3+ TFLOPS (DP) peak per chip
New AVX 512 ISA	New 512-bit Vector ISA with Masks
Scatter/Gather Engine	Hardware support for gather and scatter
New memory technology: MCDRAM + DDR	Large High Bandwidth Memory → MCDRAM Huge bulk memory → DDR
New on-die interconnect: Mesh	High BW connection between cores and memory
Integrated Fabric: Omni-Path	Better scalability to large systems. Lower Cost

Source: Avinash Sodani, Hot Chips 2015 KNL talk

KNL Performance





Significant performance improvement for compute and bandwidth sensitive workloads, while still providing good general purpose throughput performance.

1. Projected KNL Performance (1 socket, 200W CPU TDP) vs. 2 Socket Intel® Xeon® processor E5-2697v3 (2x145W CPU TDP)

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Source E5-2697v3: www.spec.org, Intel measured AlexNet Images/sec. KNL results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance. **For more information go to http://www.intel.com/performance** *Other names and brands may be claimed as the property of others.

Source: Avinash Sodani, Hot Chips 2015 KNL talk