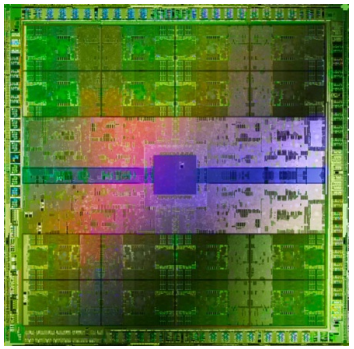
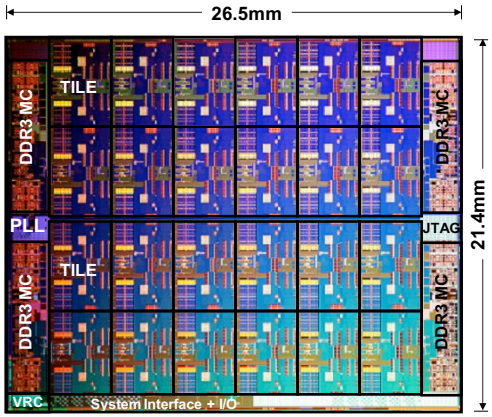
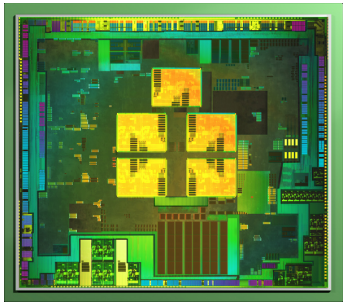


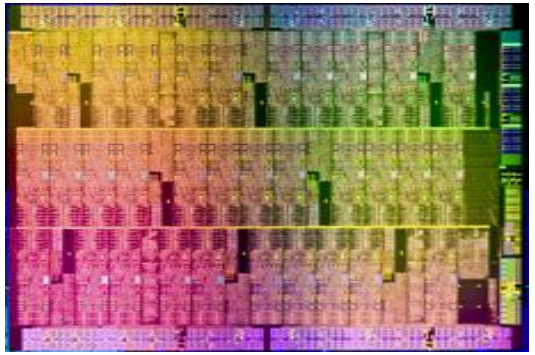
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



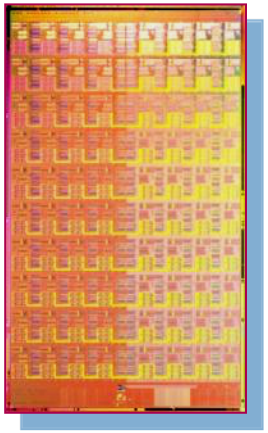
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



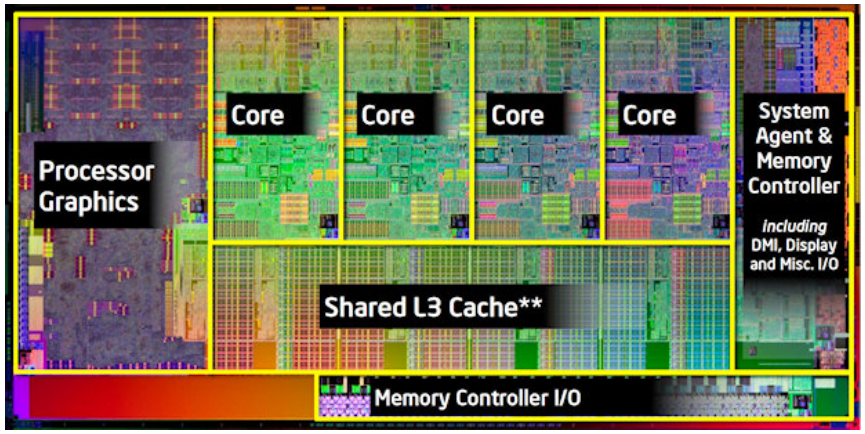
An Intel MIC processor

GPUs and the Heterogeneous programming problem

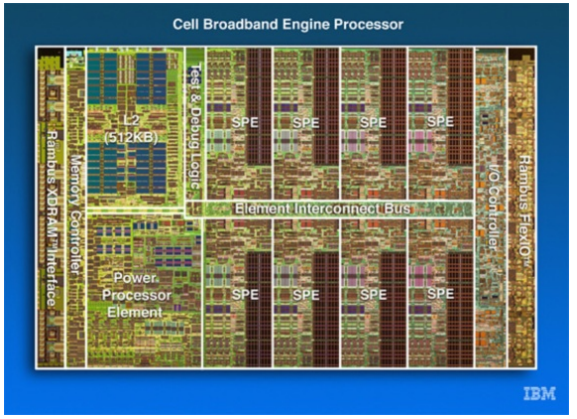
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



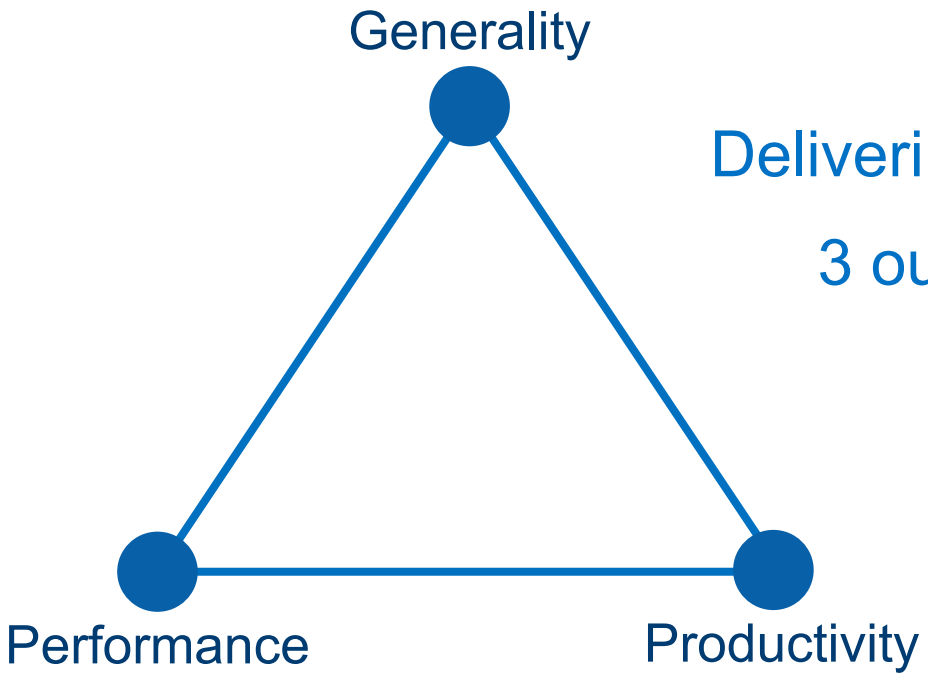
IBM Cell Broadband engine processor

Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

Outline

- Summary: programming CPUs and GPUs
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

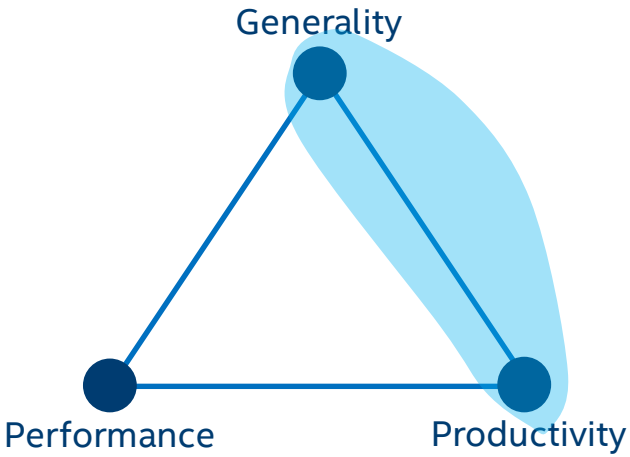
The triangle of programmer needs



Delivering 2 out of 3 is reasonable,

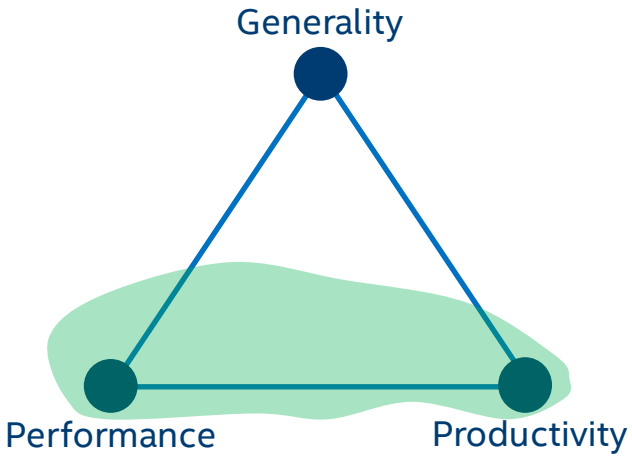
3 out of 3 is VERY difficult!!

Choice in a complex market

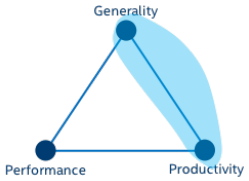


Intel chose Generality and Productivity

GPU vendors chose
Productivity and Performance

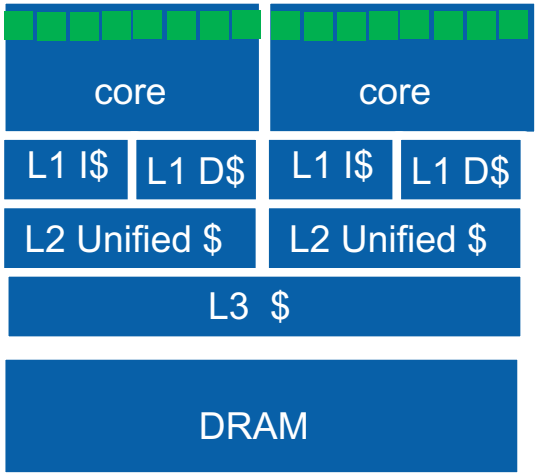


Comparing CPU and GPU programming



Programmers must master:

- Multithreading
- Complex memory hierarchy
- Vectorization

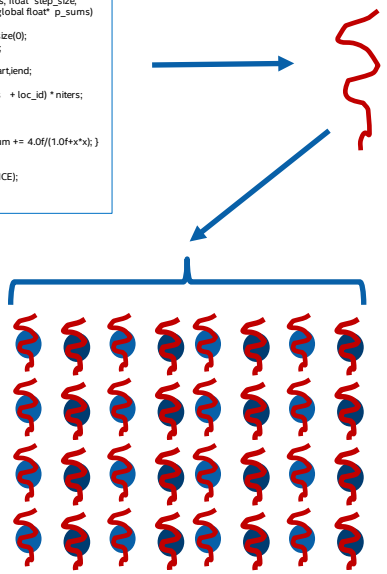


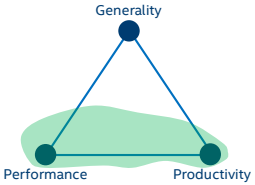
CPU: Two programming models and complex memory hierarchy.

one scalar program

```
extern void reduce(  __local float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
        __local  float*  l_sums, __global float*  p_sums)
{
  int n_wrk_items  = get_local_size(0);
  int loc_id     = get_local_id(0);
  int grp_id   = get_group_id(0);
  float x, accum = 0.0f;   int i,istart,iend;

  istart =   (grp_id * n_wrk_items    + loc_id) * niters;
  iend   = istart+niters;

  for(i= istart; i<iend; i++){
     x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

  l_sums[local_id] = accum;
  barrier(CLK_LOCAL_MEM_FENCE);
  reduce(l_sums, p_sums);
};
```

Compiled to one executable





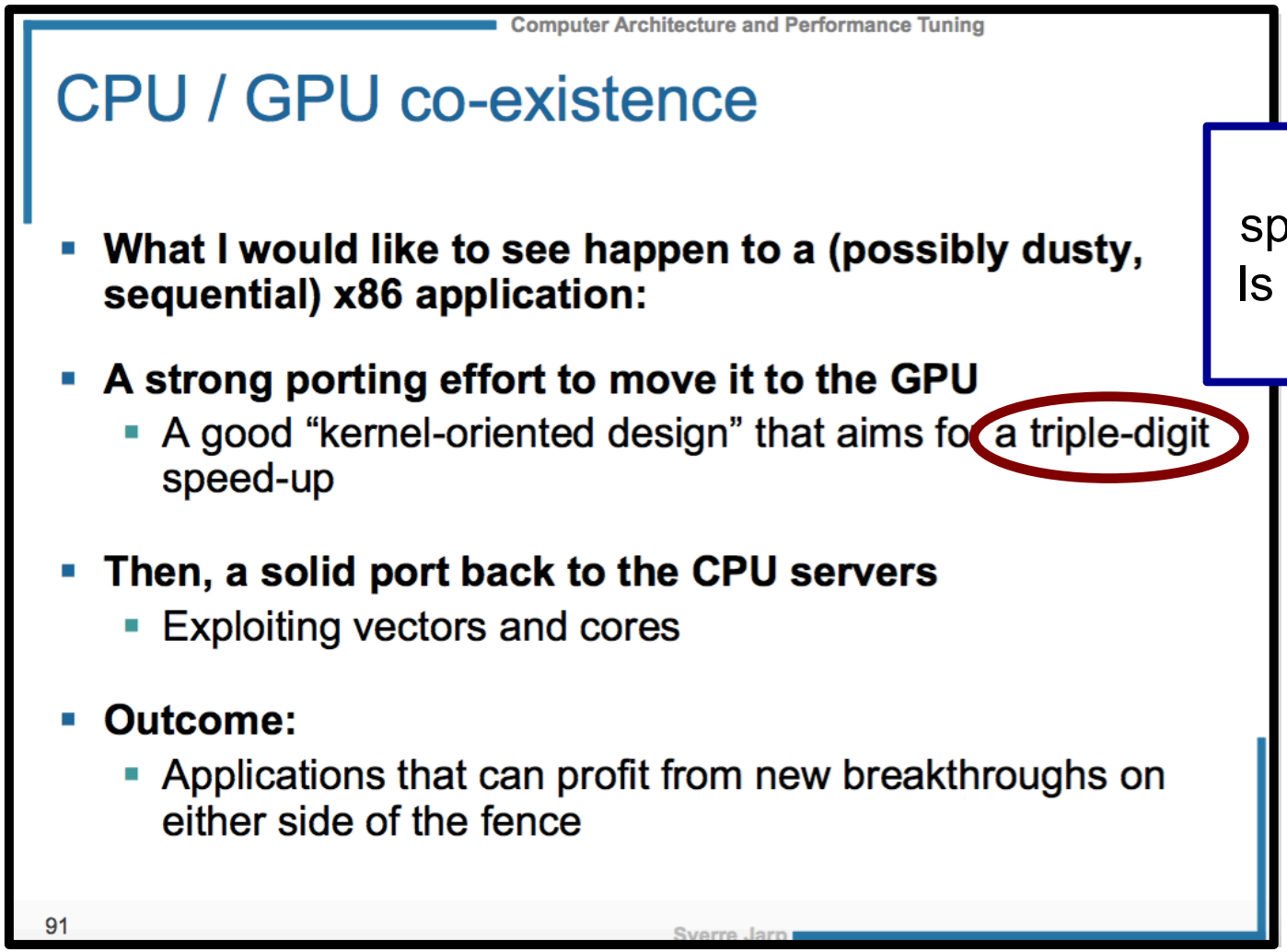
mapped onto processing elements … excess parallel work to hide complexity of memory hierarchy

GPU: One programming model (SIMT) and hidden memory hierarchy

Outline

- Summary: programming CPUs and GPUs
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

100X speedups from GPUS: a common myth



Triple digit speedups? Really? Is this a reasonable goal?

Source: a great ESC'15 lecture by a smart person who made a mistake!!!

A high level view of performance

- Well optimized applications are either compute or bandwidth bounded

- **For bandwidth bound applications:**

  *Performance = Arch efficiency * Peak Bandwidth Capability*

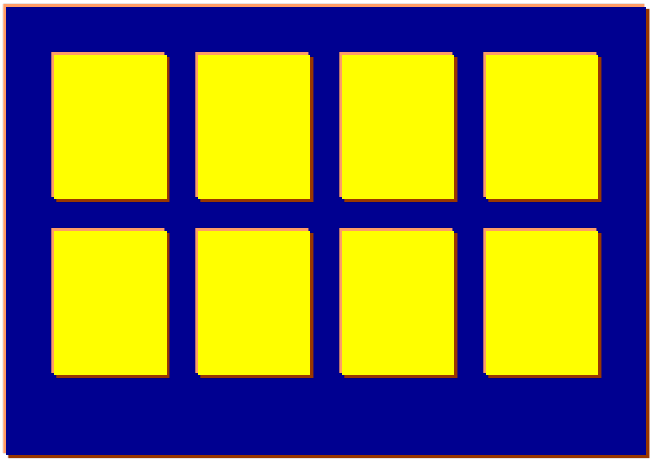
- **For compute bound applications:**

  *Performance = Arch efficiency * Peak Compute Capability*

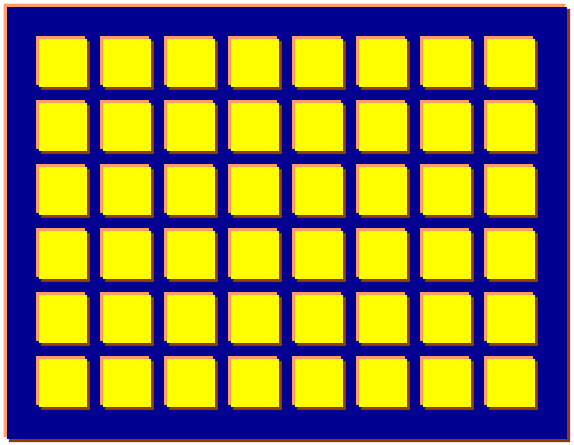
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Reasonable Speedup Expectations

- Chip A



- Chip B



$Perf_A = Eff_A * Peak_A(\text{Comp or BW})$

$Perf_B = Eff_B * Peak_B(\text{Comp or BW})$

$$Speedup\frac{B}{A} = \frac{Perf_B}{Perf_A} = \frac{Eff_B}{Eff_A} * \frac{Peak_A(Comp\_or\_BW)}{Peak_B(Comp\_or\_BW)}$$

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

Speedup expectations for well optimized code: CPU vs. GPU

Core i7 960

- Four OoO Superscalar Cores, 3.2GHz
- Peak SP Flop: 102GF/s
- Peak BW: 30 GB/s

GTX 280

- 30 SMs (w/ 8 In-order SP each), 1.3GHz
- Peak SP Flop: 933GF/s*
- Peak BW: 141 GB/s

Assuming both Core i7 and GTX280 have the same efficiency:

	Max Speedup: GTX 280 over Core i7 960
Compute Bound Apps: (SP)	933/102 = 9.1x
Bandwidth Bound Apps:	141/30 = 4.7x

* 933GF/s assumes mul-add and the use of SFU every cycle on GPU

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

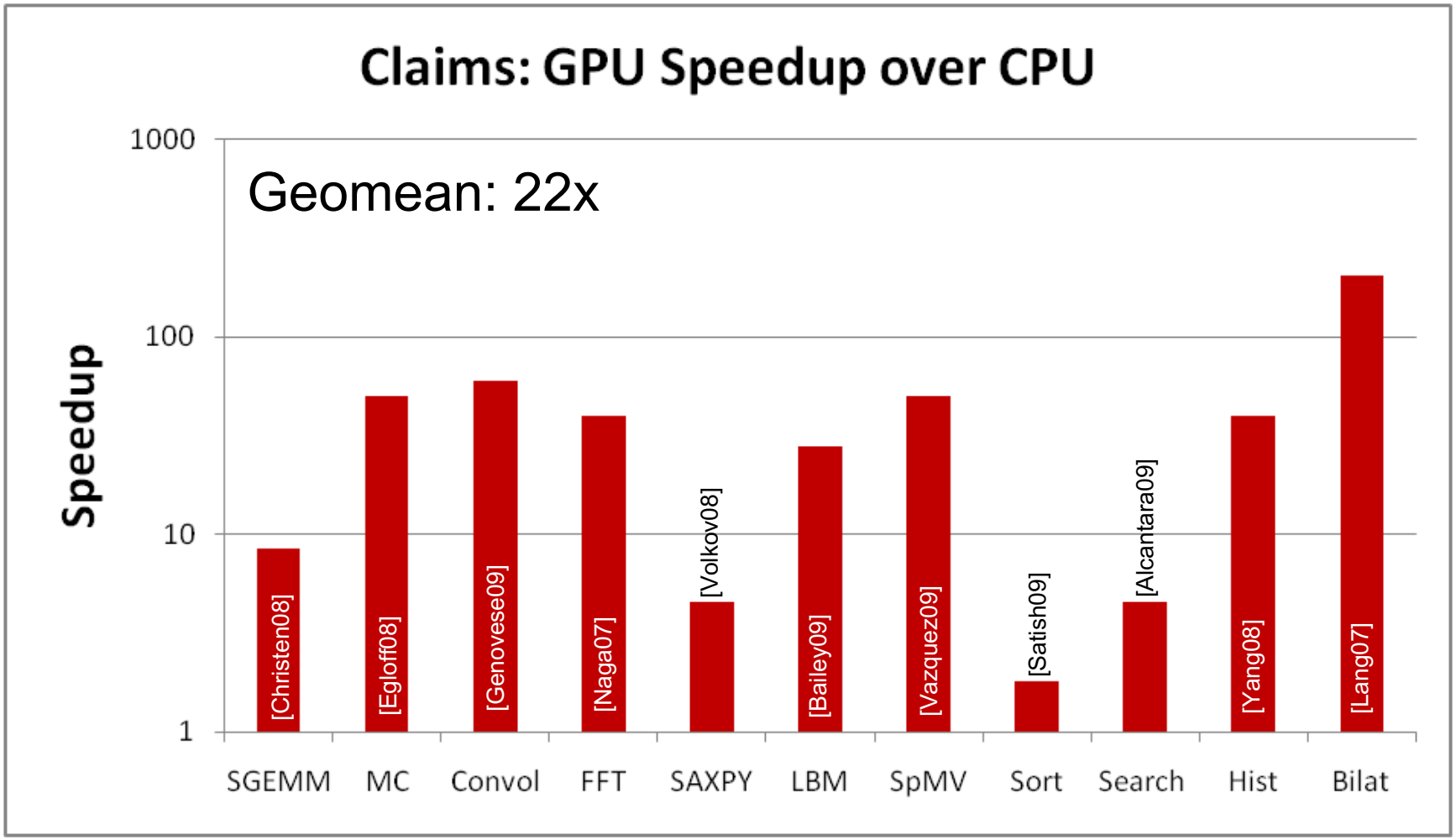
A fair comparison of CPUs and GPUs: Methodology

- 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

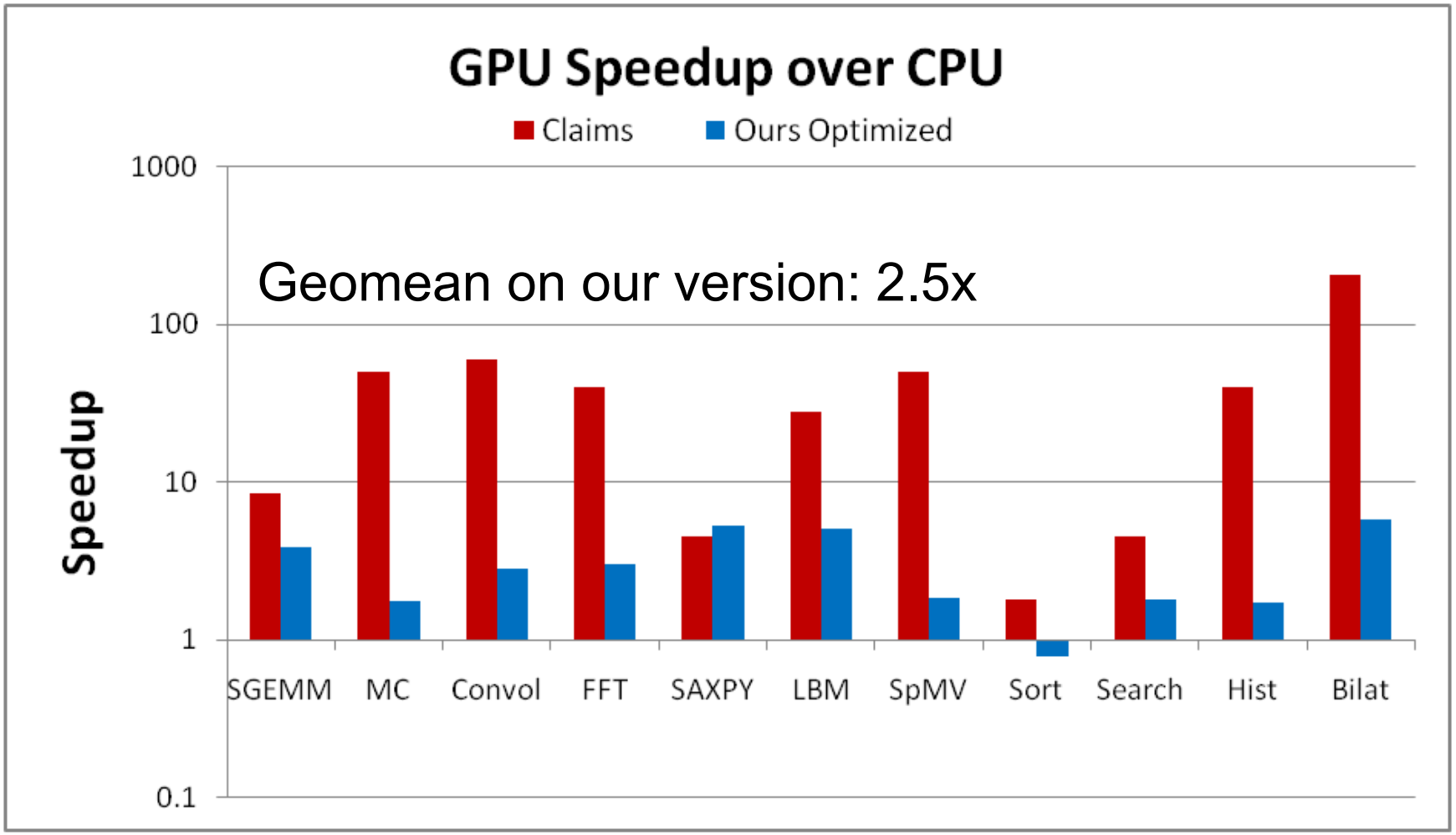
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What was claimed



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

What we measured

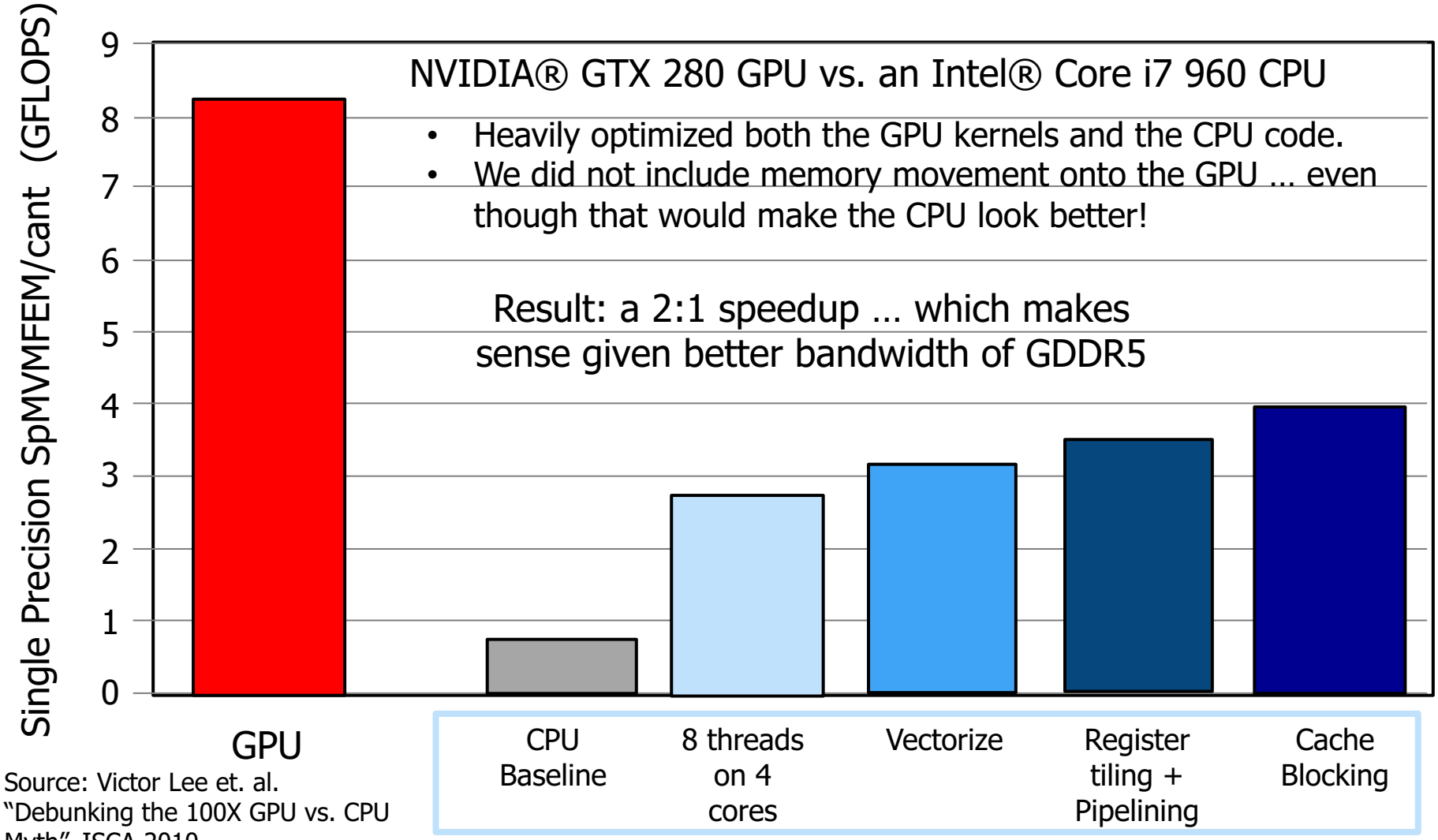


Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

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

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

*third party names are the property of their owners

Common Mistakes when comparing a CPU and a GPU

- Compare the latest GPU against an old CPU
- Highly optimized GPU code vs. unoptimized CPU code
  - I've seen numerous papers compare optimized CUDA vs. Matlab or python
- Parallel GPU code vs. serial, unvectorized CPU code.
- Ignore the GPU penalty of moving data across the PCI bus from the CPU to the GPU

GPUs are great and depending on the algorithm can show two to four fold speedups. But not 100+ … that's just irresponsible and should not be tolerated!!

Outline

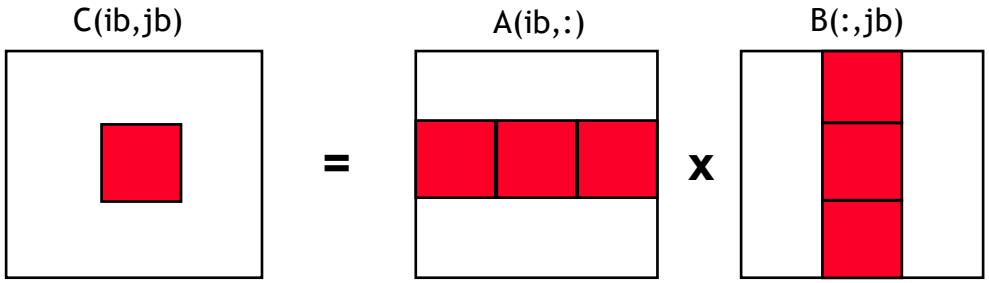
- Summary: programming CPUs and GPUs
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

Whining about performance Portability

- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It’s called OpenCL

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;

  // 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;
}
```

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

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			

* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Xeon Phi™ SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB

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.

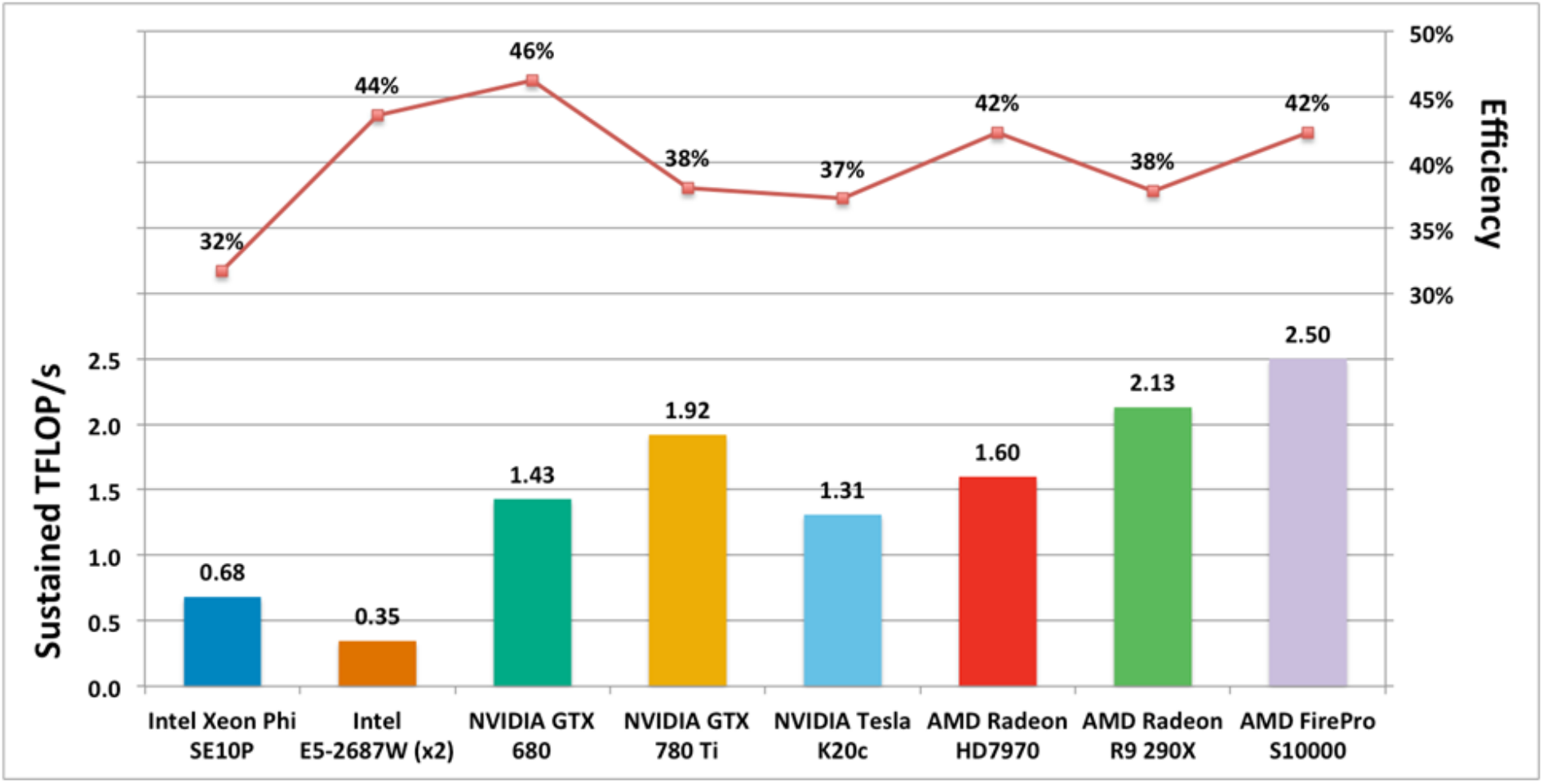
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

Third party names are the property of their owners.

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



Whining about performance Portability

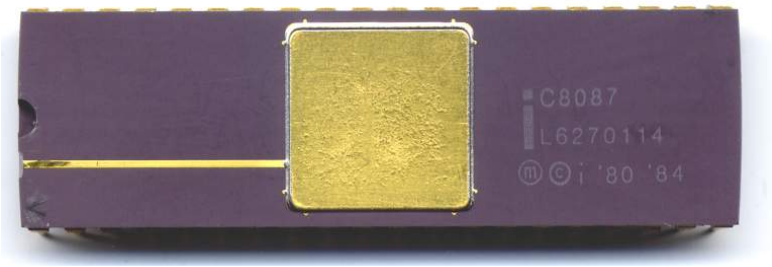
- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It’s called OpenCL
- But this focus on mythical “Performance Portability” misses the point. The issue is “maintainability”.
  - You must be able maintain a body of code that will live for many years over many different systems.
  - Having a common code base using a portable programming environment … even if you must fill the code with if-defs or have architecture specific versions of key kernels … is the only way to support maintainability.

Outline

- Summary: programming CPUs and GPUs
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

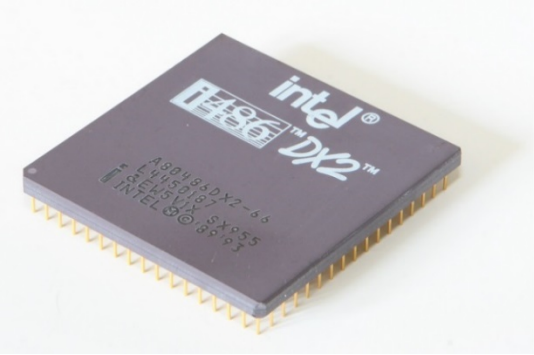
Coprocessors to accelerate flops

- 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.

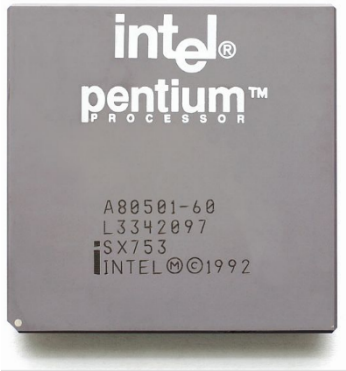


- Related Standards:
  - Partnership between industry and academia led to IEEE 754 …. The most important standard in the history of HPC. IEEE 754 first supported by x87.

- 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.

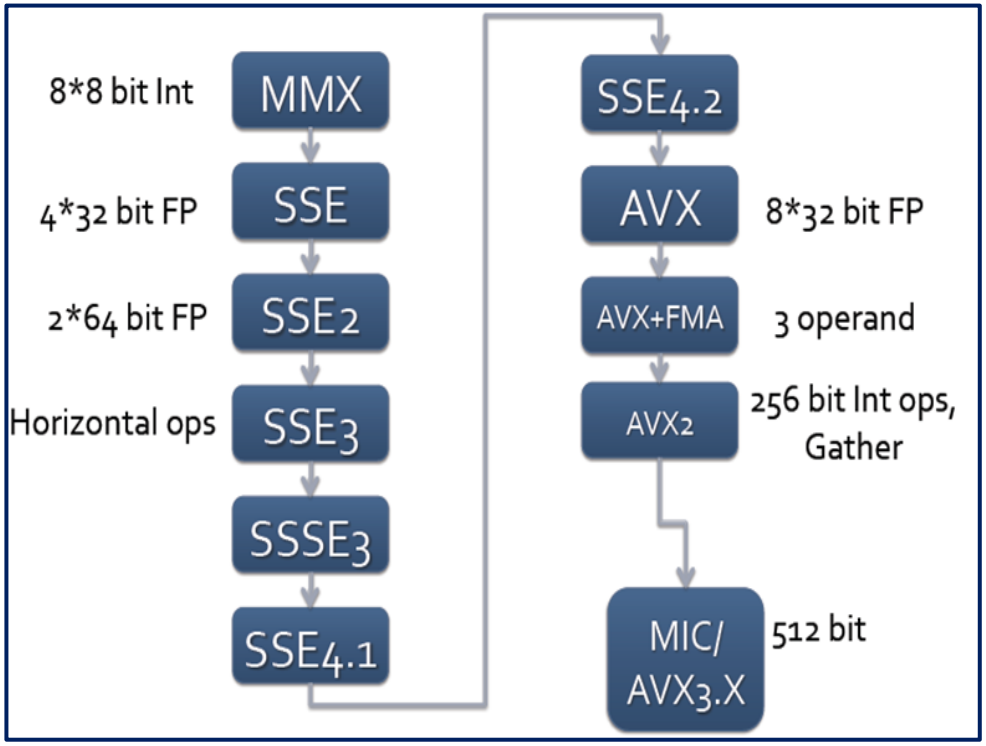
Intel C8087 Math Coprocessor. Source: http://www.cpu-collection.de/?l0=i&i=1698&sd=1

Vector processing accelerators

- Vector Coprocessor:
  - Vector co-processor (from Sky computer) for Intel iPSC/2 MPP (~1987)
  - Floating point systems array processors (late 80s's)

Absorbed into the CPU

The Intel® i860 processor (early 90's) with integrated vector instructions.

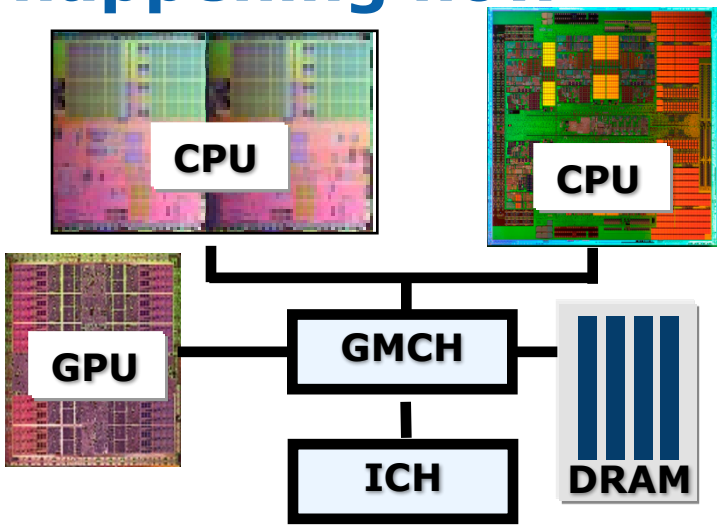


And now vector instructions are a ubiquitous element of CPUs.

*third party names are the property of their owners

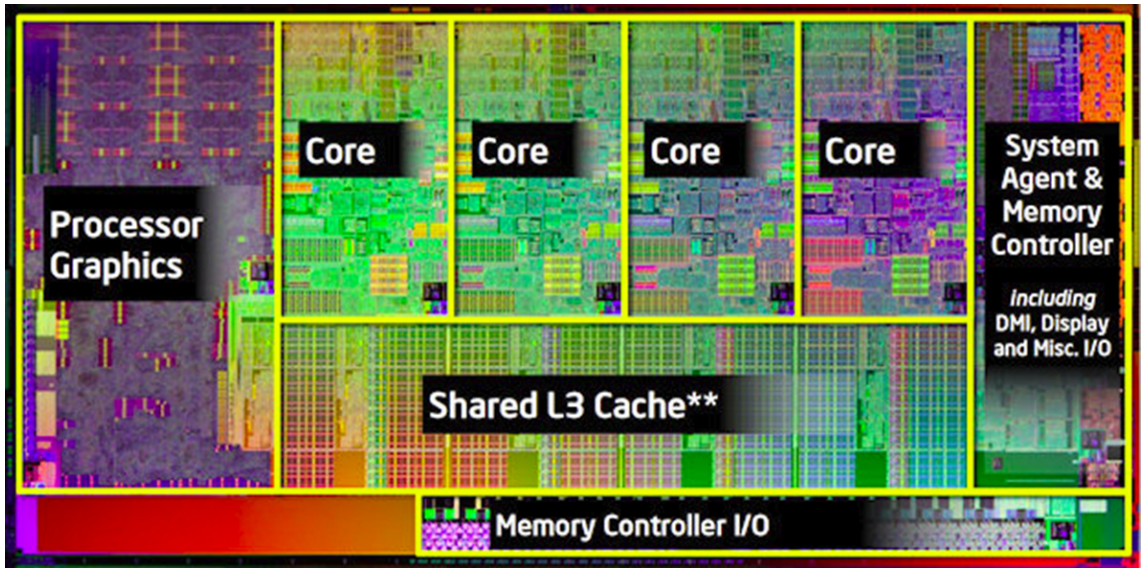
Source: http://www.cpu-collection.de/?l0=co&l1=Intel&l2=i860

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

Conclusion

- The SIMT platform is here to stay … though the GPU is likely to move from a discrete card to an IP block on the CPU die.
- Performance benefits are significant (two to four times) but not silly (100+).
- The more interesting question …
  - Is the SIMT platform as a software abstraction better than multi-threading + vectorization?
  - The corporate CPU world says “no” but they are not the final judges of this fundamental programmability question. It’s up to applications programmers to decide.
  - So try both and see what you think … threads+vectorization or SIMT?