2494-17

Workshop on High Performance Computing (HPC) Architecture and Applications in the ICTP

14 - 25 October 2013

Eurotech company overview and HPC division

G. Tecchiolli
Eurotech, Udine

EUROTECH

Imagine. Build. Succeed.

Workshop on HPC Architecture and Applications
ICTP - Trieste

Giampietro Tecchiolli

Trieste - 2013 10 21

Agenda

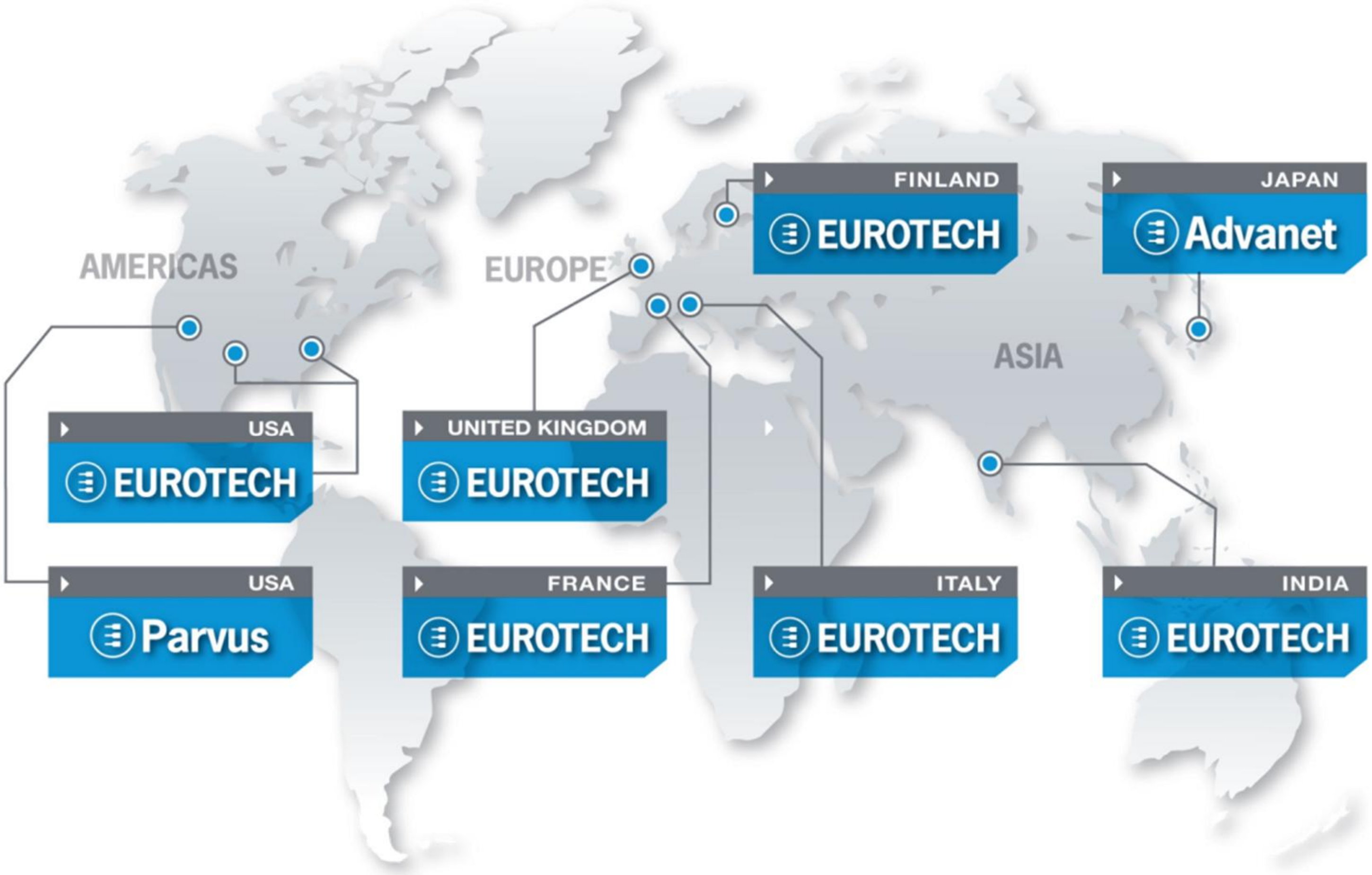
- **Introduction**
- **Beyond the Moore's Law**
  - MIC Computing
  - GPU Computing
- **Energy-Efficient Computing**
  - RISC Architectures (ARM)
  - Reconfigurable Architectures (FPGA)

Eurotech company overview and HPC division

EUROTECH

Eurotech Introduction

Group Global Footprint



HPC division highlights

- The Eurotech HPC division focuses on **designing, manufacturing, delivering and supporting high performance computing solutions**
- More than 14 years of history of delivering supercomputing systems and solutions to industry and academia
- First worldwide company to market hot water cooled high performance computers. First hot water cooled HPC in the market delivered in 2009.
- R&D capabilities nurtured in house and through collaboration with the best universities and research centres in Europe: INFN, Julich, Revensburg, Daisy…
- Funding member of ETP for HPC

ETP 4 HPC

Eurotech HPC project examples

Ape mille, 1999-2002
Ape next, 2002-2005

Janus, 2006-2008

Q-Pace, 2007-2009

Aurora Science, 2008-2010

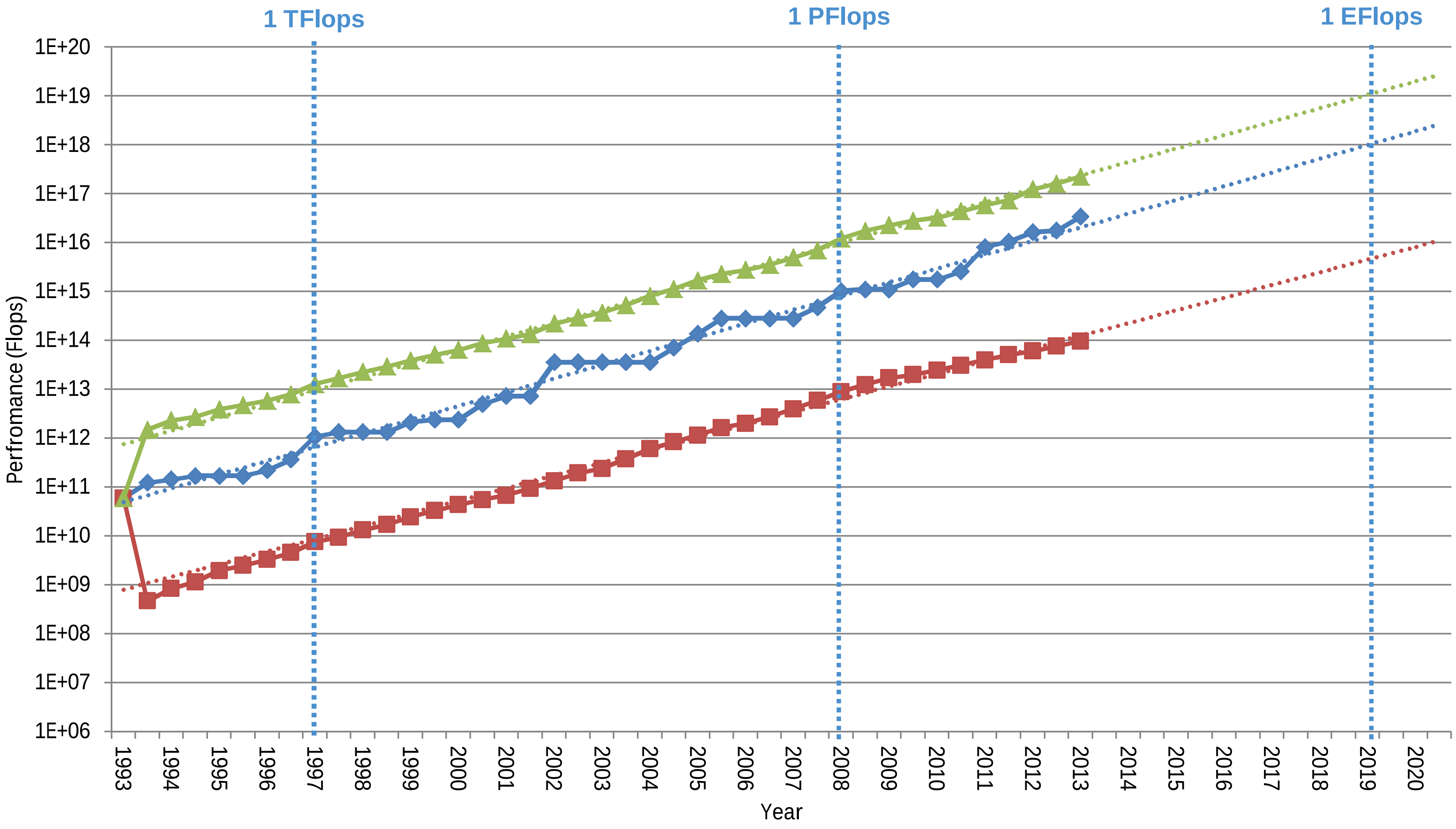
Eurora 2012

Deep project 2012

Selex Elsag (E-security) 2011-2012

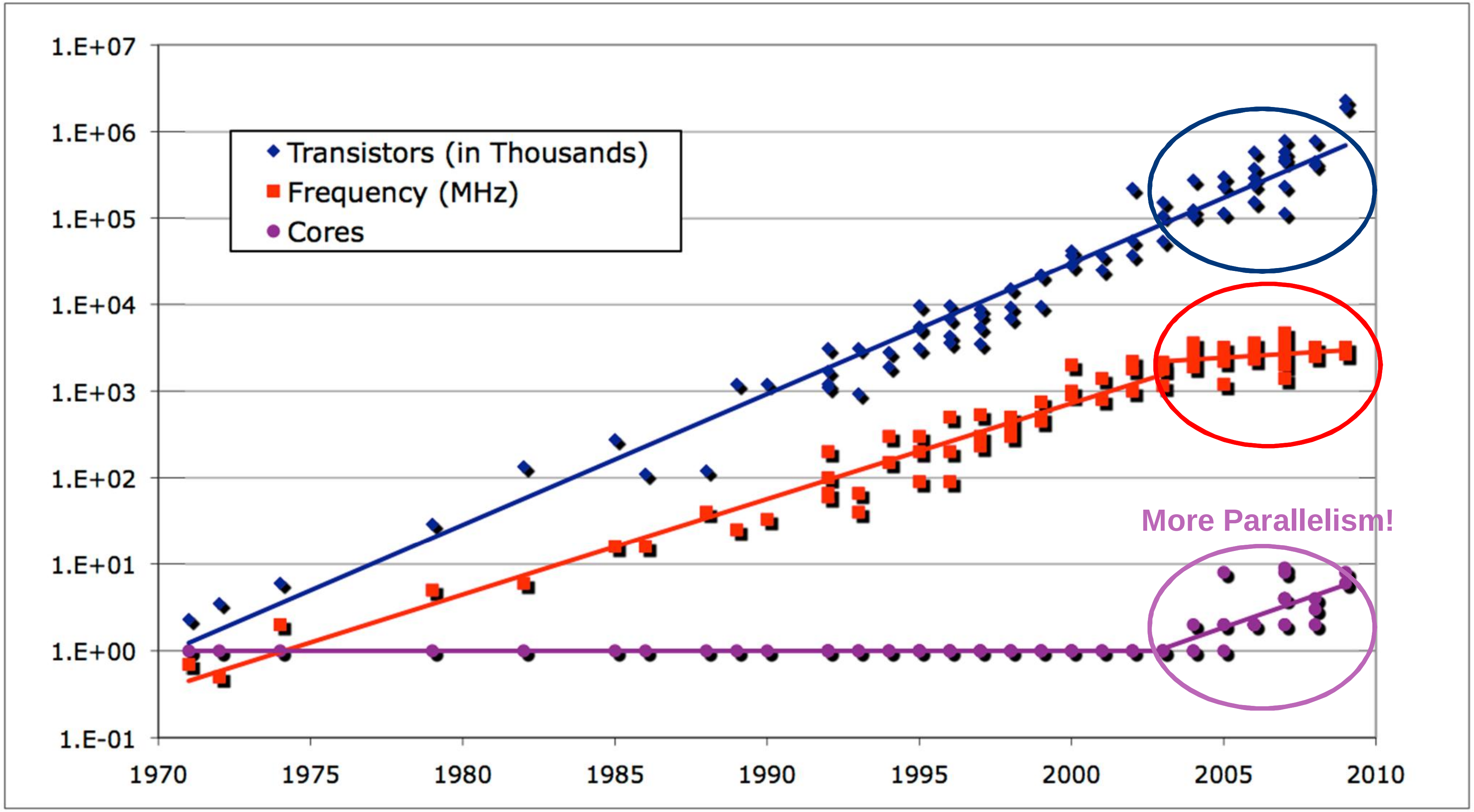
TRENDS IN HPC

Top500 trend



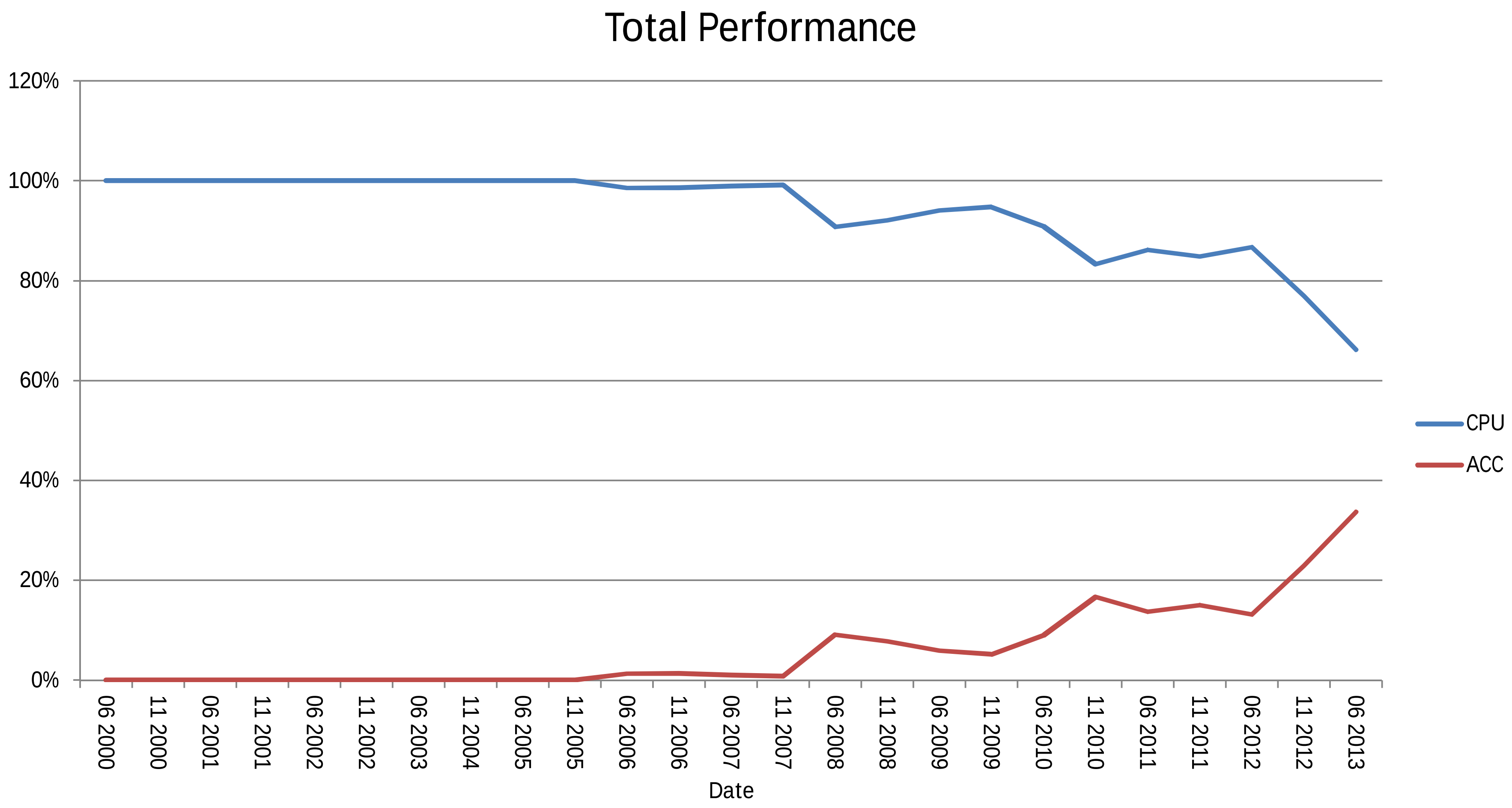
EUROTECH

HPC Trend and Moore’s Law



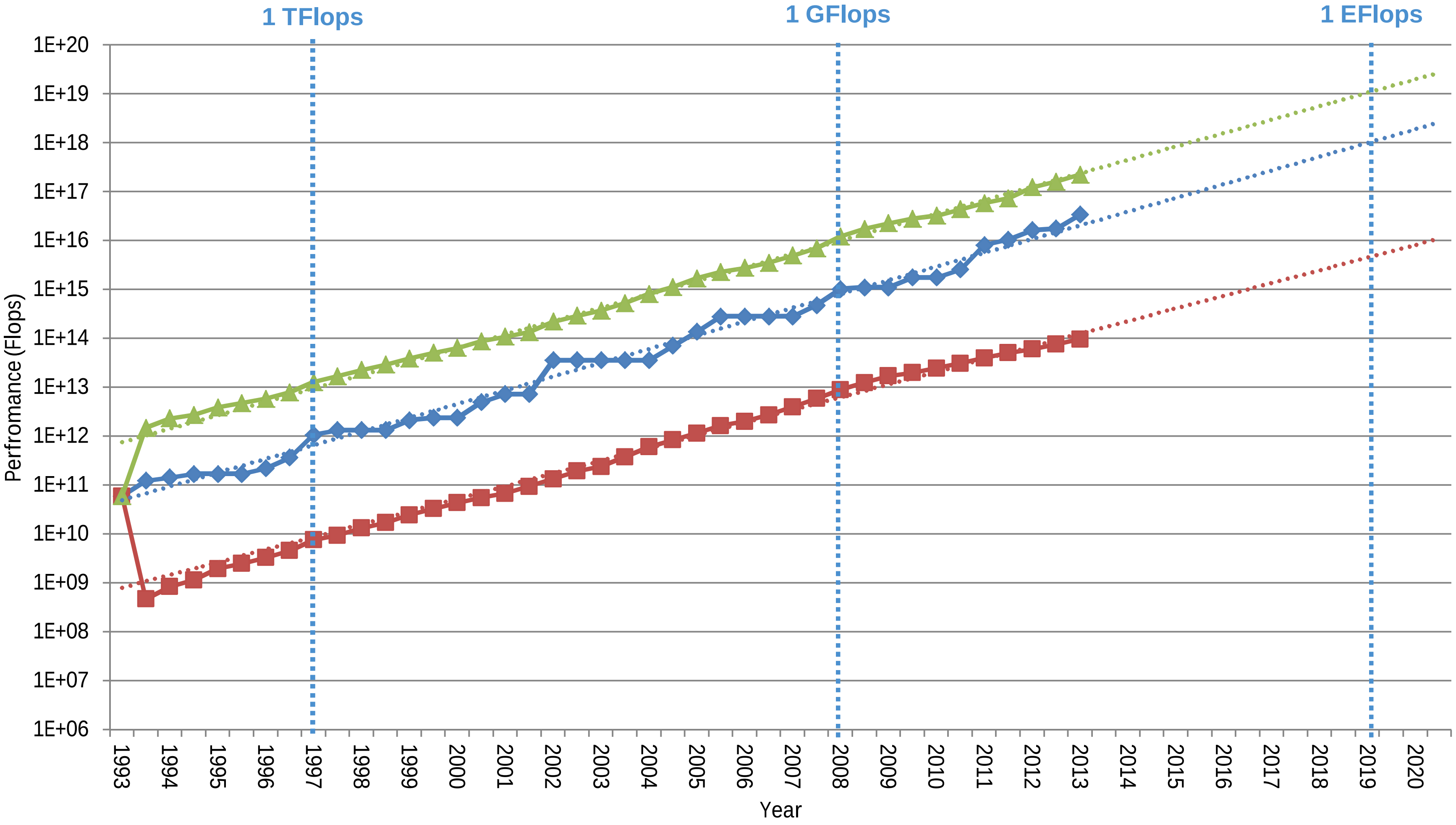
Scalar (CPU) vs Accelerated (ACC) computing

(i.e. Beyond the Moore's Law)



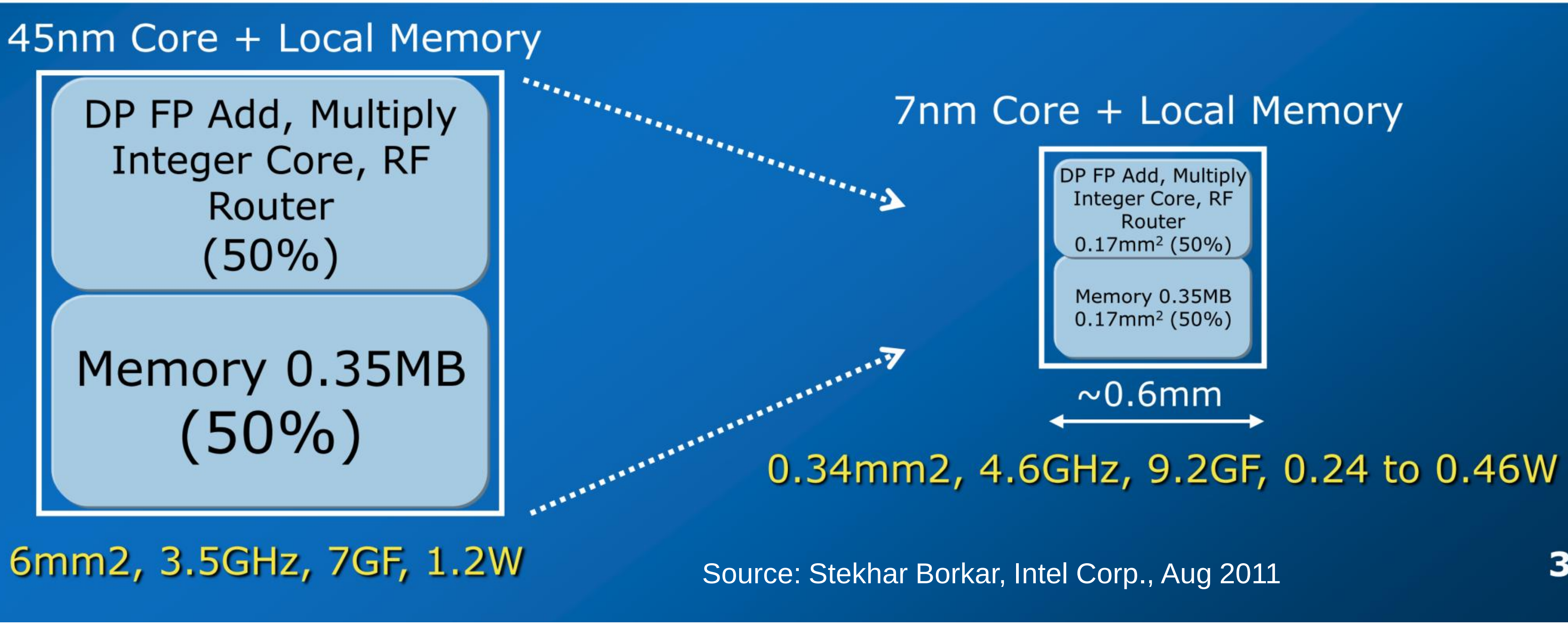
EUROTECH

Top500 trend



Technology Outlook

Technology (High Volume)	45nm (2008)	32nm (2010)	22nm (2012)	14nm (2014)	10nm (2016)	7nm (2018)	5nm (2020)
Transistor density	1.75	1.75	1.75	1.75	1.75	1.75	1.75
Frequency scaling	15%	10%	8%	5%	4%	3%	2%
Vdd scaling	-10%	-7.5%	-5%	-2.5%	-1.5%	-1%	-0.5%
Dimension & Capacitance	0.75	0.75	0.75	0.75	0.75	0.75	0.75
SD Leakage scaling/micron	1X Optimistic to 1.43X Pessimistic						



45nm Core + Local Memory

DP FP Add, Multiply Integer Core, RF Router (50%)

Memory 0.35MB (50%)

6mm2, 3.5GHz, 7GF, 1.2W

7nm Core + Local Memory

DP FP Add, Multiply Integer Core, RF Router 0.17mm^2 (50%)

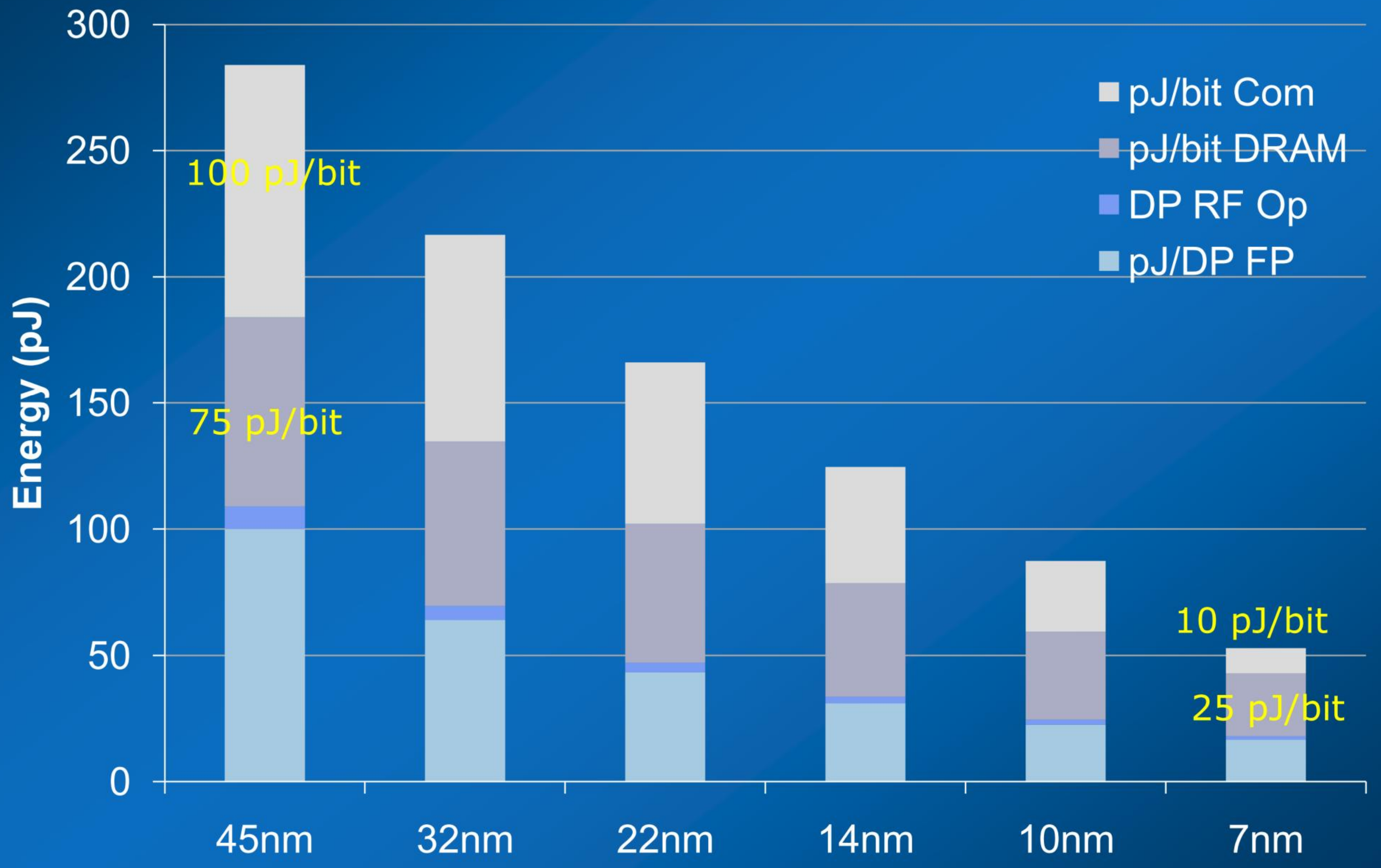
Memory 0.35MB 0.17mm^2 (50%)

~0.6mm

0.34mm2, 4.6GHz, 9.2GF, 0.24 to 0.46W

Source: Stekhar Borkar, Intel Corp., Aug 2011

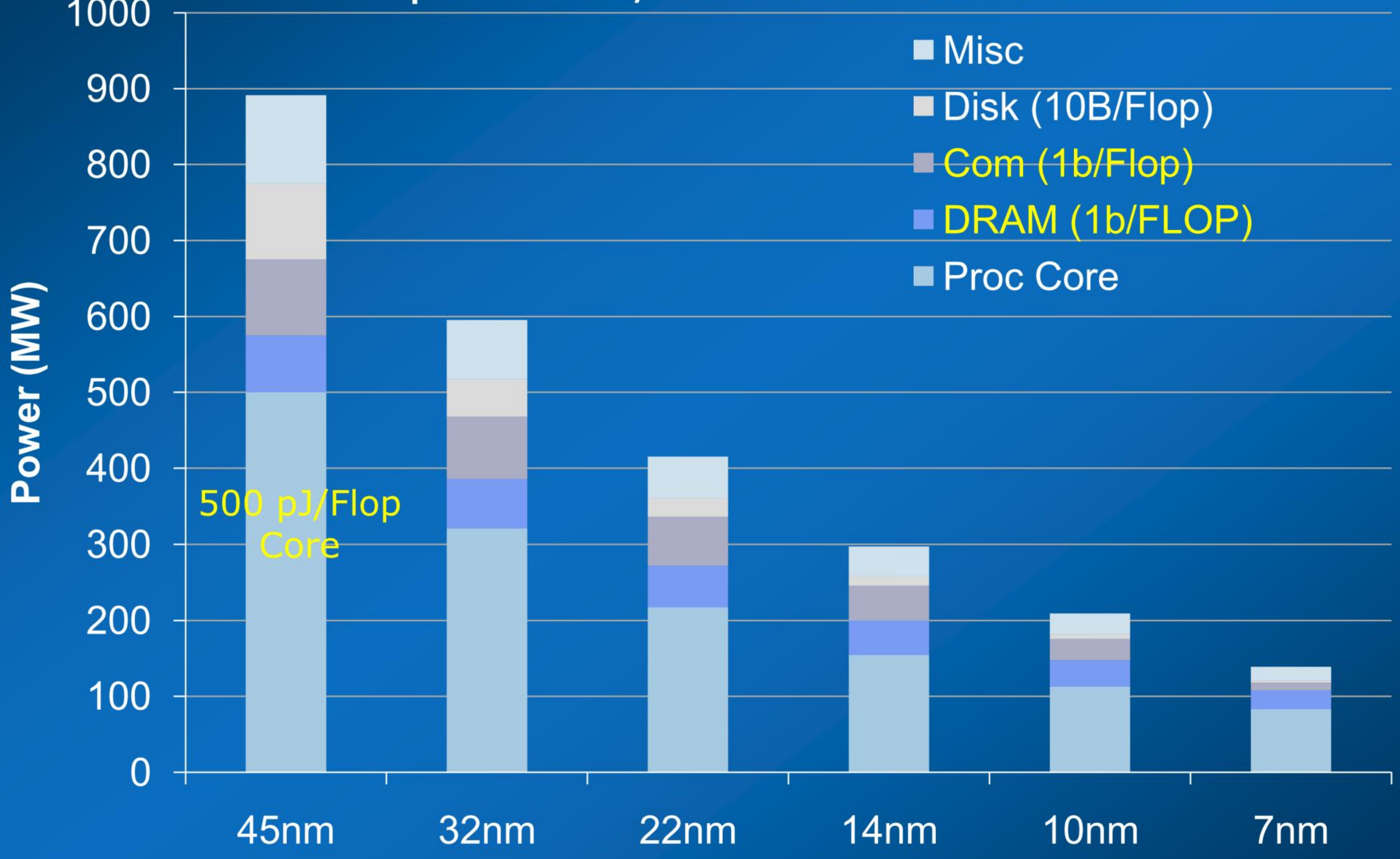
Energy per Operation



Source: Stekhar Borkar, Intel Corp., Aug 2011

Peak Exascale System Power

Optimistic, **with** constraints

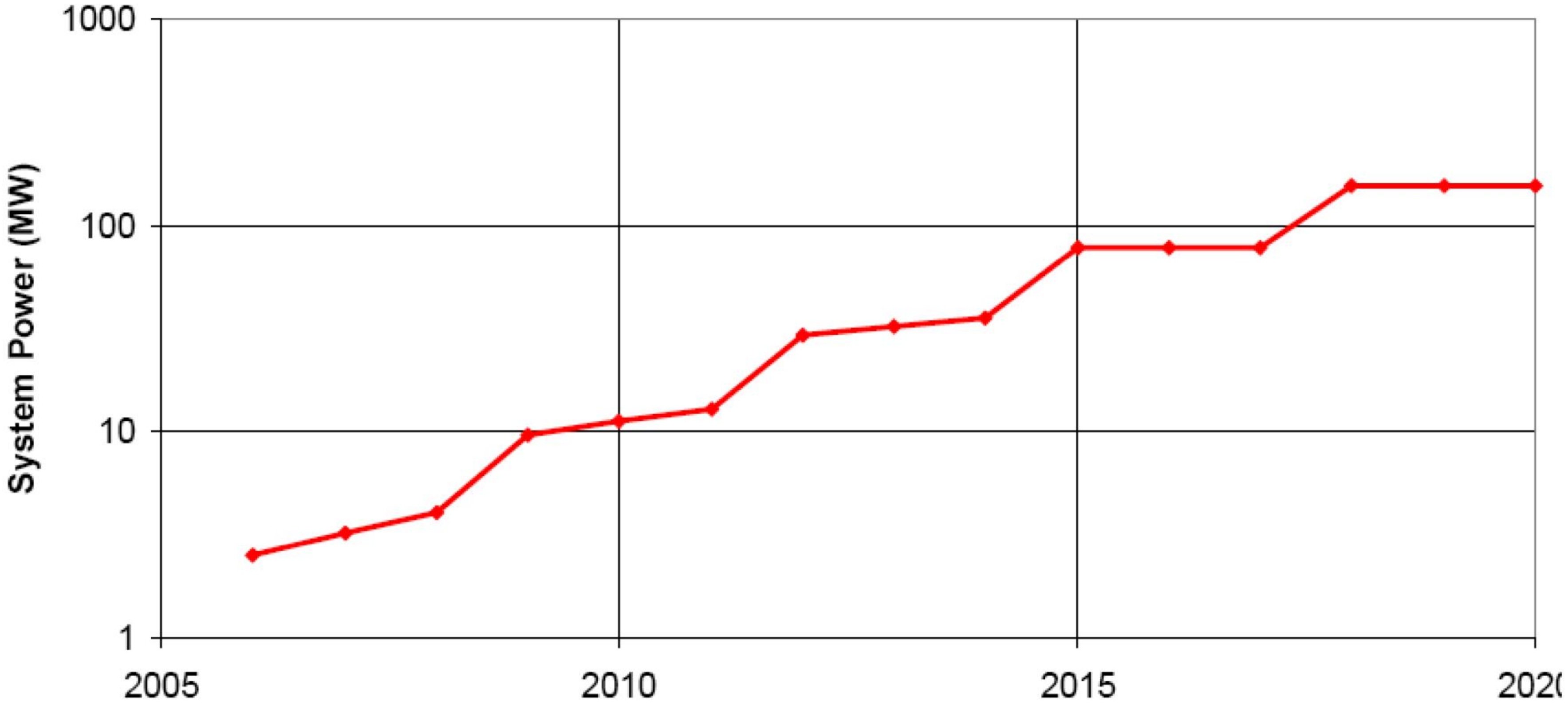


500 pJ/Flop Core

Source: Stekhar Borkar, Intel Corp., Aug 2011

Darpa Study 2008

(i.e. more Energy-Efficient Computing)

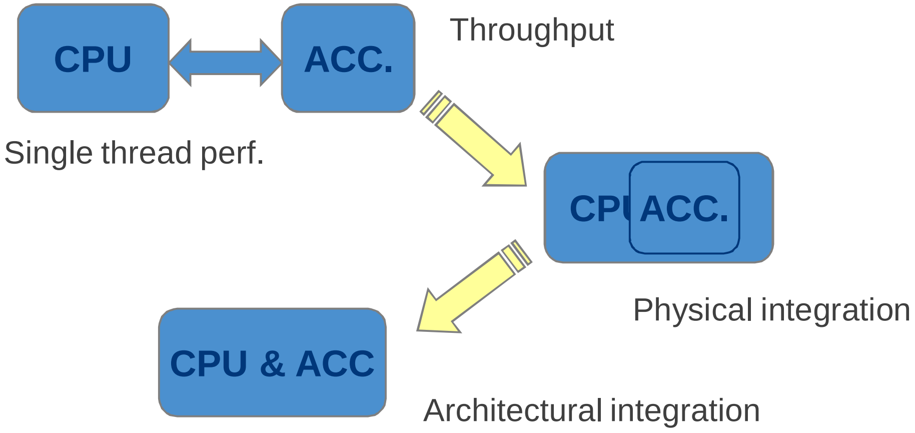


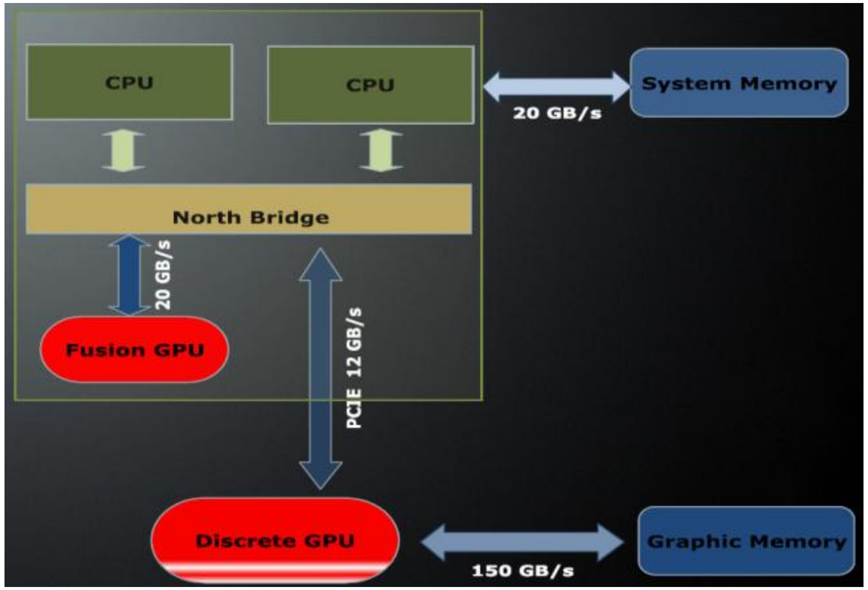
From Peter Kogge,
DARPA Exascale Study

ACCELERATED COMPUTING

What is an Accelerator.

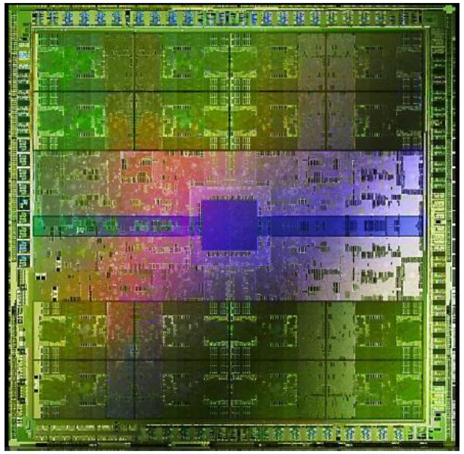
- A set (one or more) of very simple execution units that can perform few operations (with respect to standard CPU) with very high efficiency. When combined with full featured CPU (CISC or RISC) can accelerate the “nominal” speed of a system.





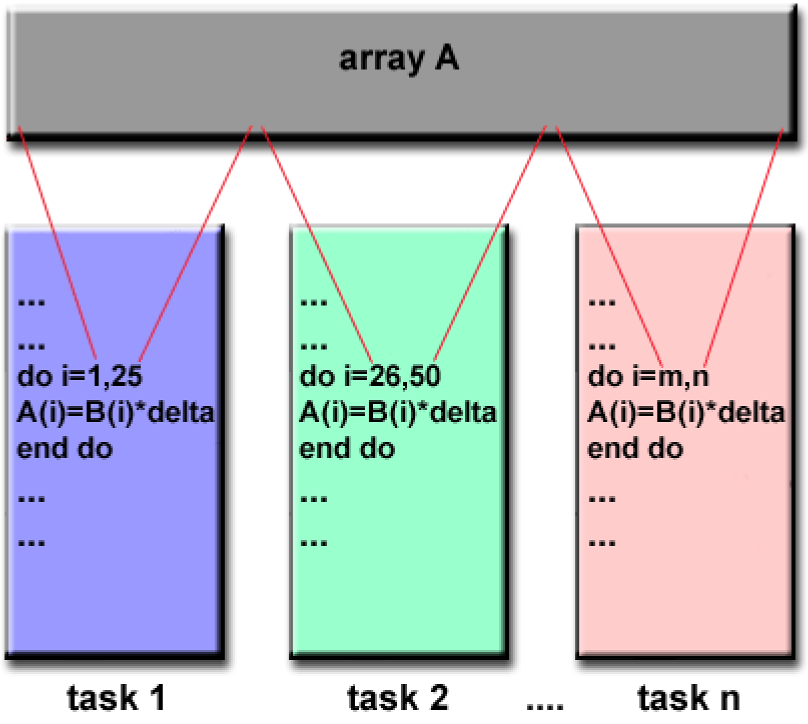
- Main approaches to accelerators:
  - Task Parallelism (MIMD) → MIC
  - Data Parallelism (SIMD) → GPU
  - Reconfigurable Devices

GPU



DATA Parallelism (SIMD)

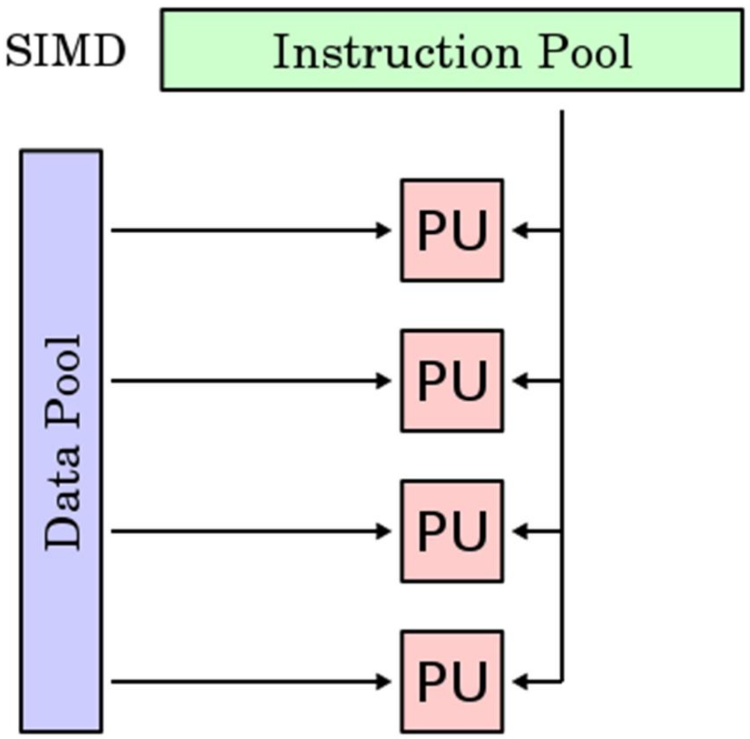
```
do i=1,10000
          A(i) = B(i) * delta
end do
```



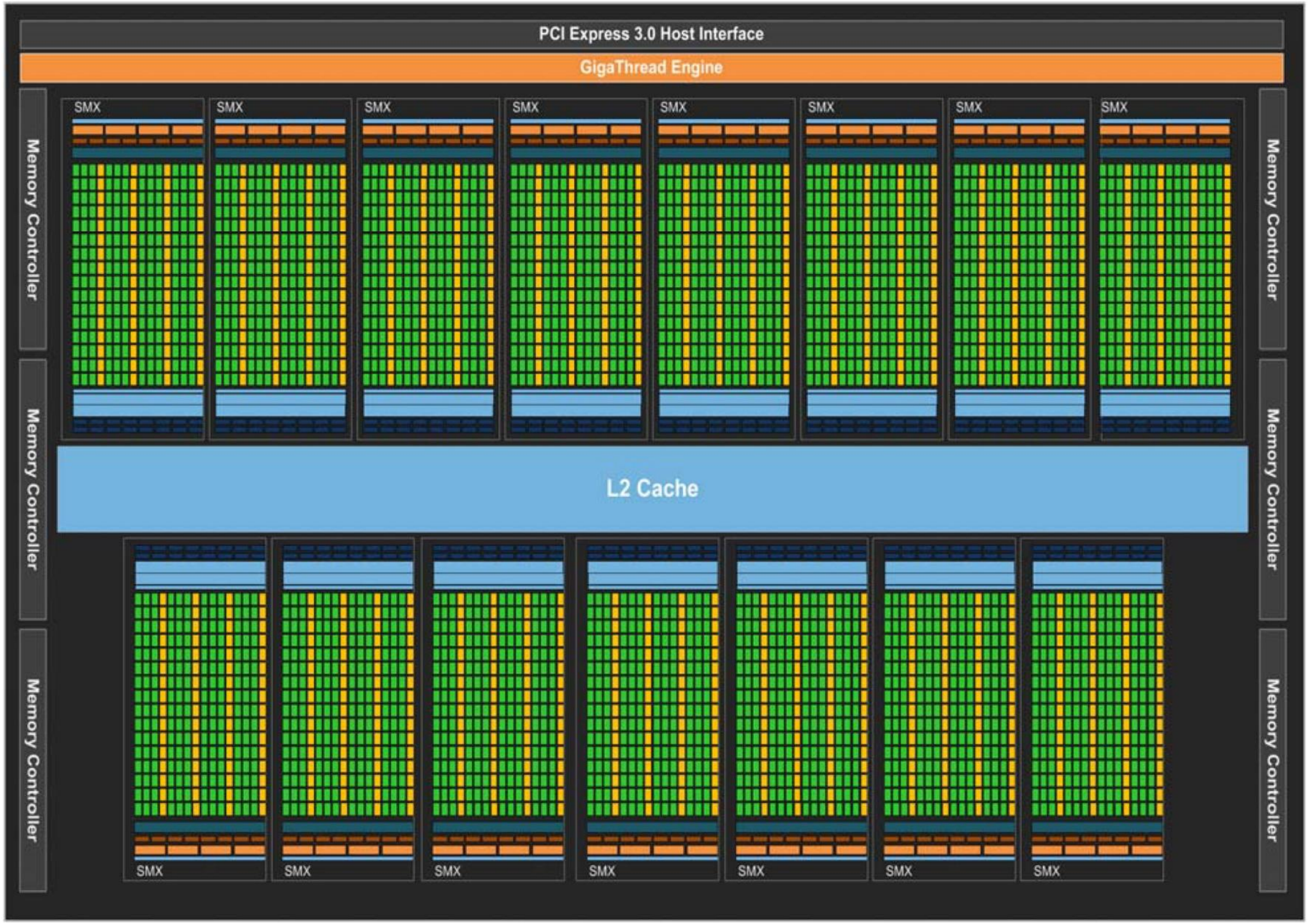
```
...
...
do i=1,25
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=26,50
A(i)=B(i)*delta
end do
...
...
```

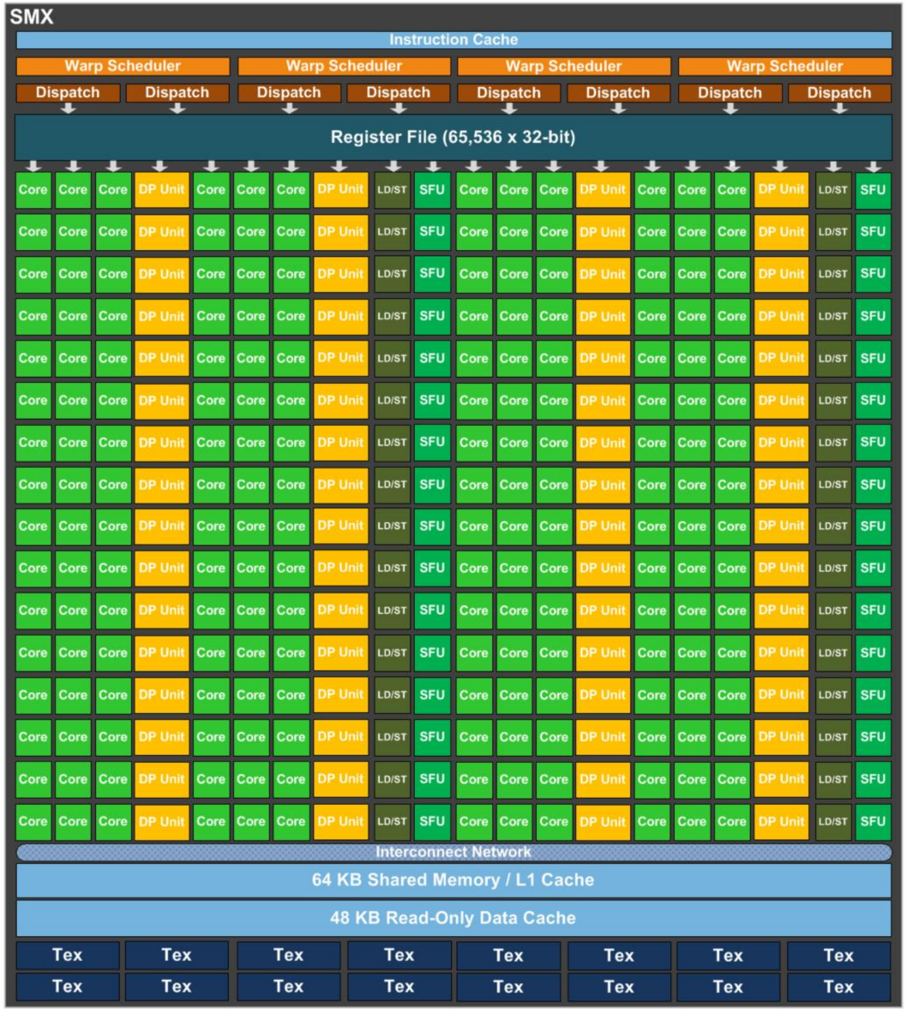
```
...
...
do i=m,n
A(i)=B(i)*delta
end do
...
...
```

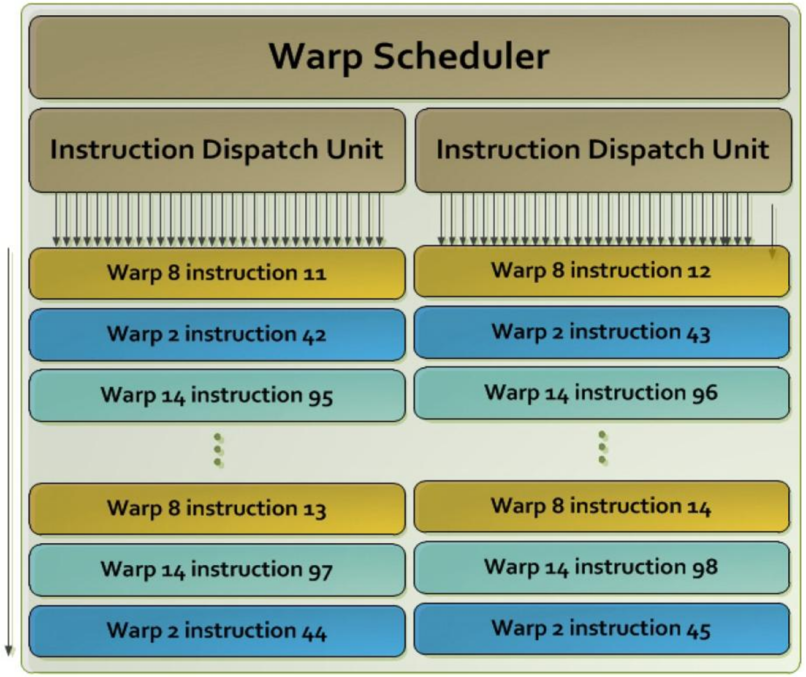


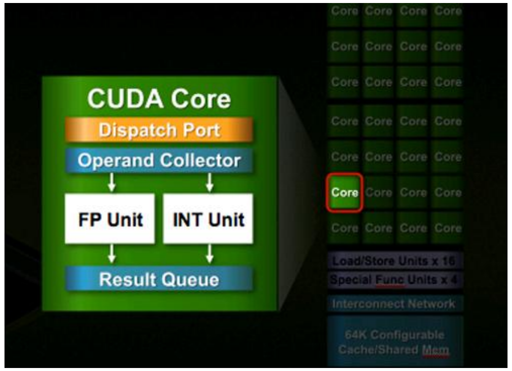
GigaThread Engine (Grid)



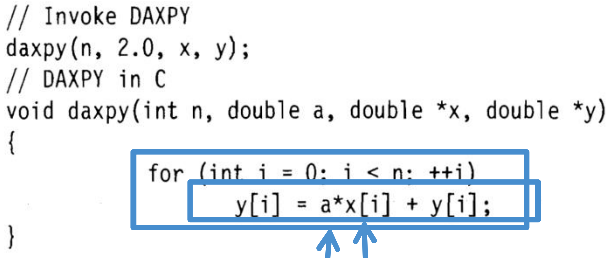
SMX Processor & Warp Scheduler & Core







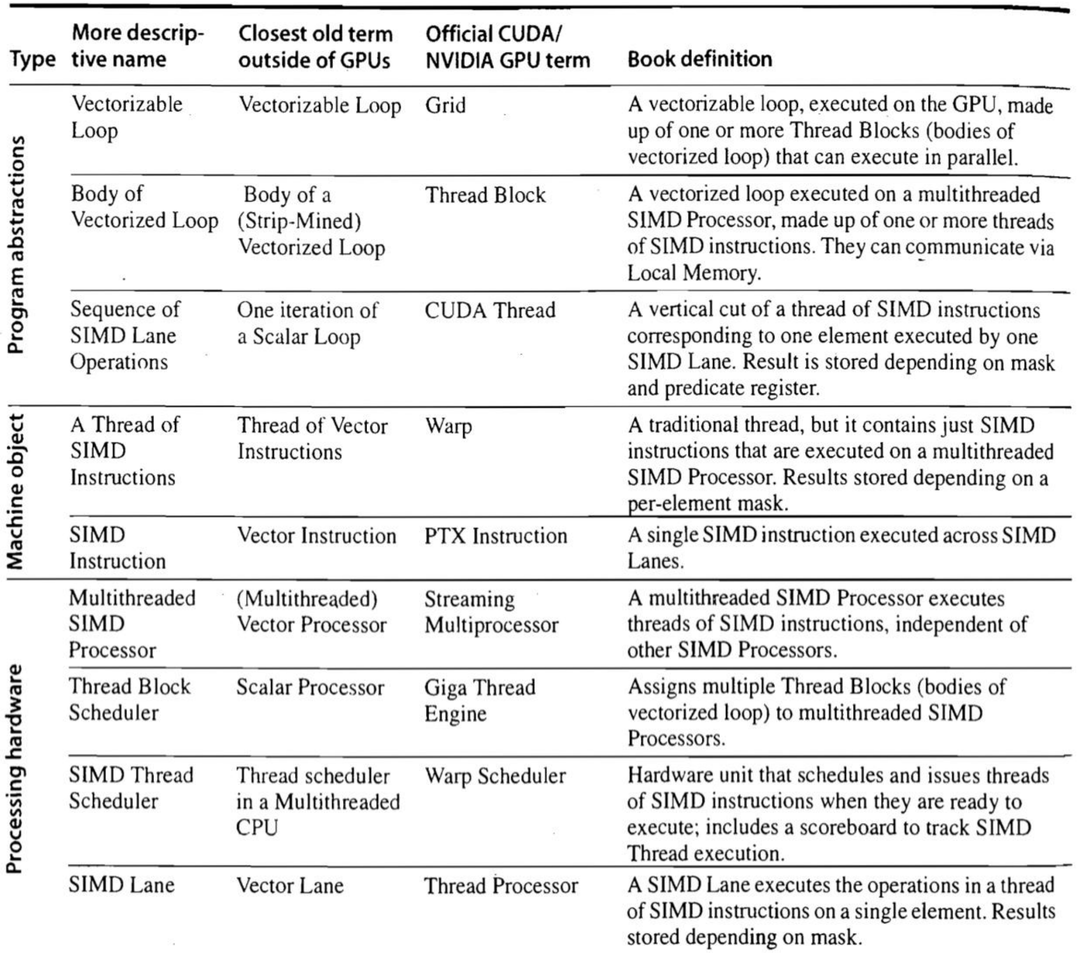
DAXPY (Y ← a * X + Y) in action



```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
          for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

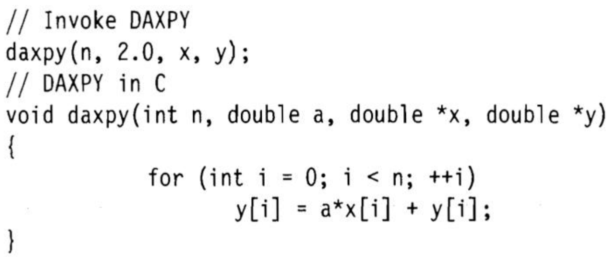
1. A vectorizable (i.e. data parallel) loop is managed at **Grid** Level (pool of threads)
2. The body of a vectorizable loop is managed by a **thread block** an managed by the SMX processors
3. Each **thread** is managed by the warp scheduler that dynamically issues the PTX instruxctions in a pool of excution units (cores)

y[18] = a*x[18] + y[18]

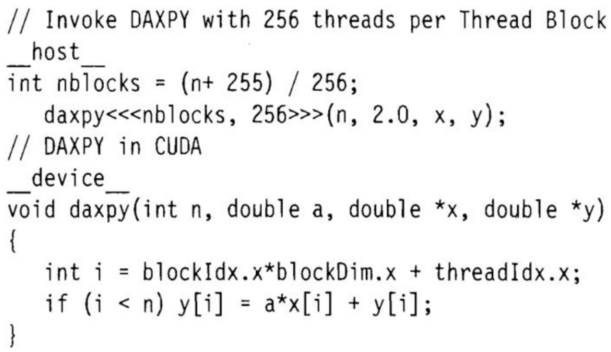


Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.

DAXPY (Y ← a * X + Y) in CUDA



```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
          for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

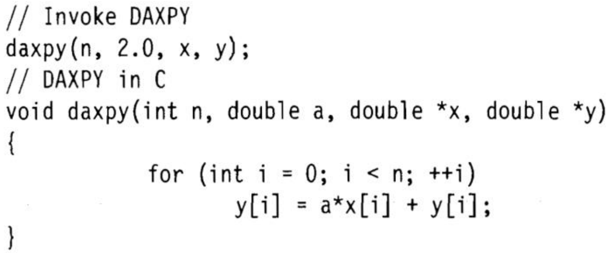


```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

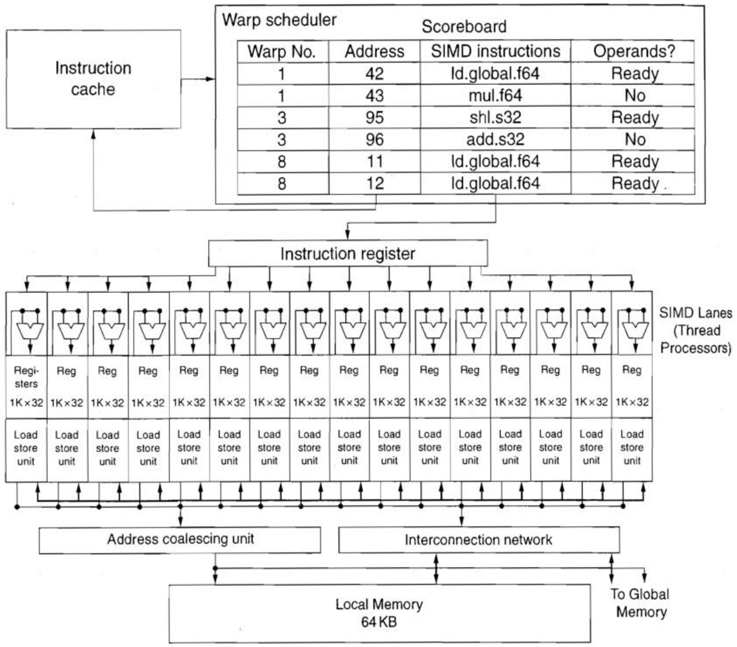
1. A vectorizable (i.e. data parallel) loop is managed at Grid Level (pool of threads)
2. The body of a vectorizable loop is managed by a thread block an managed at SMX level
3. Each thread is managed by the warp scheduler that dynamically issues the PTX instruxctions in a pool of excution units (cores)

- __***host***__ instructs the compiler to generate code and to allocate data in the CPU scope
- __***device***__ instructs the compiler to generate code and to allocate data in the GPU scope

DAXPY Summary



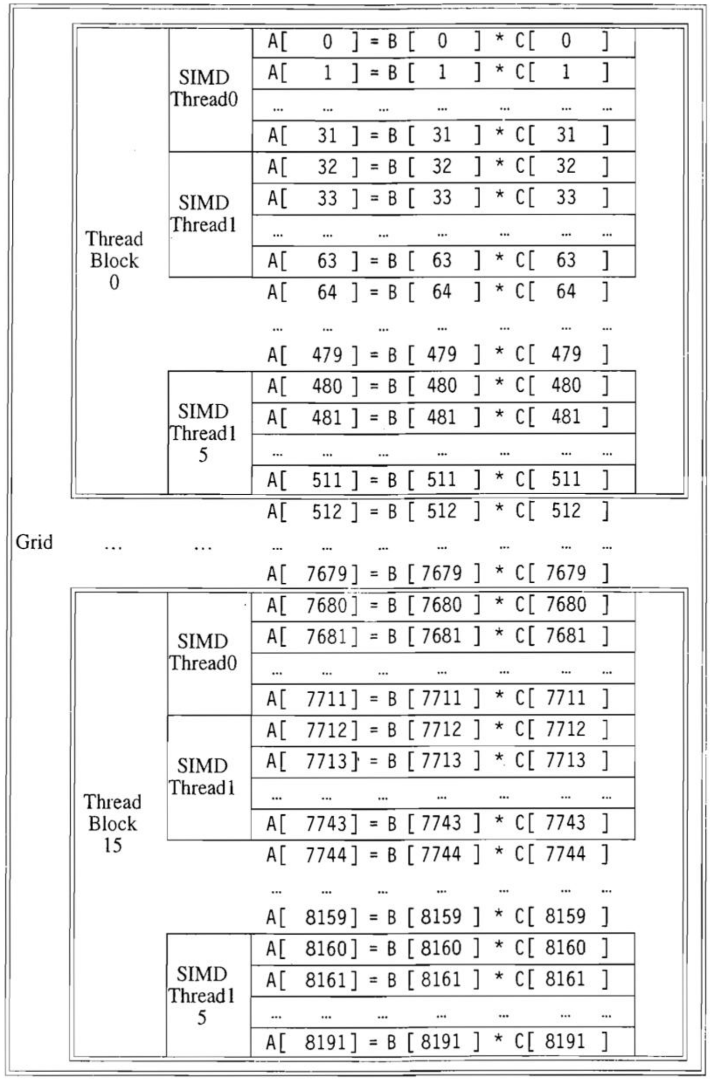
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
            for (int i = 0; i < n; ++i)
                    y[i] = a*x[i] + y[i];
}
```



Warp scheduler

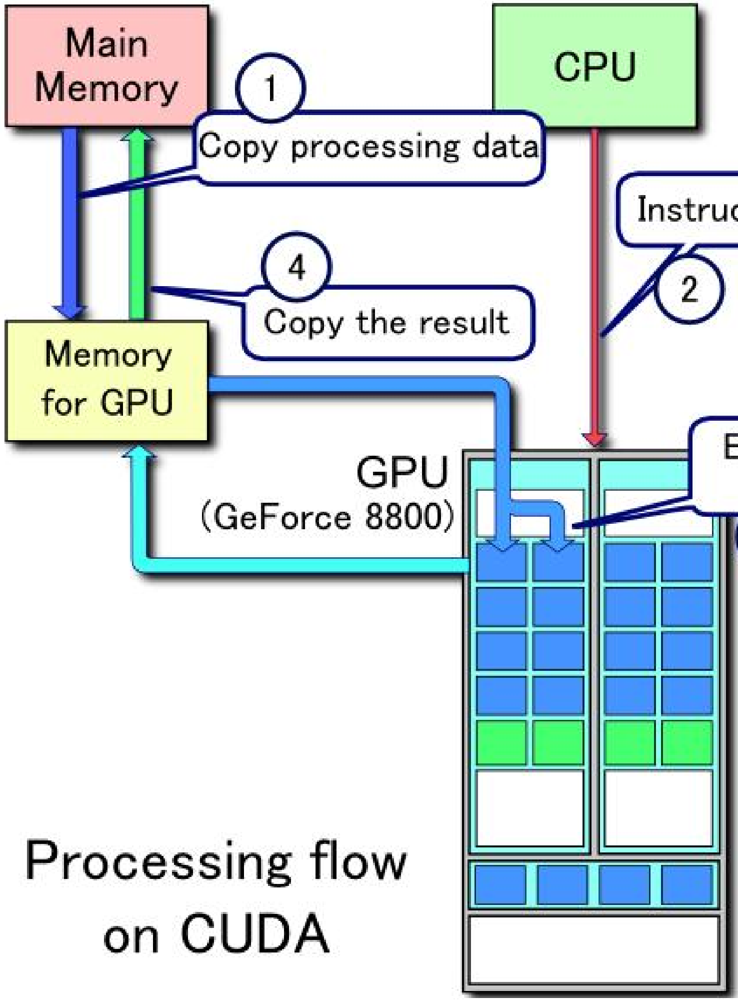
Scoreboard

Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

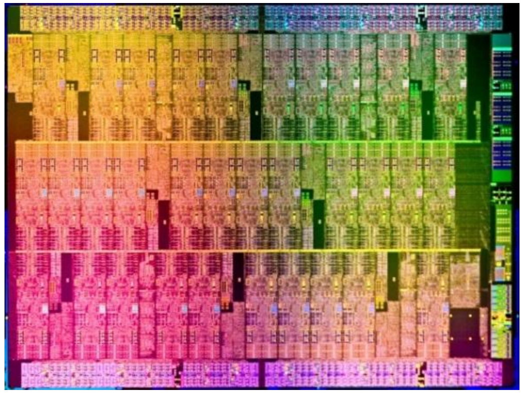


Grid	Thread Block 0	SIMD Thread0	A[0] = B [0] * C[0]
			A[1] = B [1] * C[1]
			... ...
			A[31] = B [31] * C[31]
		SIMD Thread1	A[32] = B [32] * C[32]
			A[33] = B [33] * C[33]
			... ...
			A[63] = B [63] * C[63]
			A[64] = B [64] * C[64]
			... ...
			A[479] = B [479] * C[479]
		SIMD Thread15	A[480] = B [480] * C[480]
			A[481] = B [481] * C[481]
			... ...
			A[511] = B [511] * C[511]
			A[512] = B [512] * C[512]
	...	...	... ...
			A[7679] = B [7679] * C[7679]
	Thread Block 15	SIMD Thread0	A[7680] = B [7680] * C[7680]
			A[7681] = B [7681] * C[7681]
			... ...
			A[7711] = B [7711] * C[7711]
		SIMD Thread1	A[7712] = B [7712] * C[7712]
			A[7713] = B [7713] * C[7713]
			... ...
			A[7743] = B [7743] * C[7743]
			A[7744] = B [7744] * C[7744]
			... ...
			A[8159] = B [8159] * C[8159]
		SIMD Thread15	A[8160] = B [8160] * C[8160]
			A[8161] = B [8161] * C[8161]
			... ...
			A[8191] = B [8191] * C[8191]

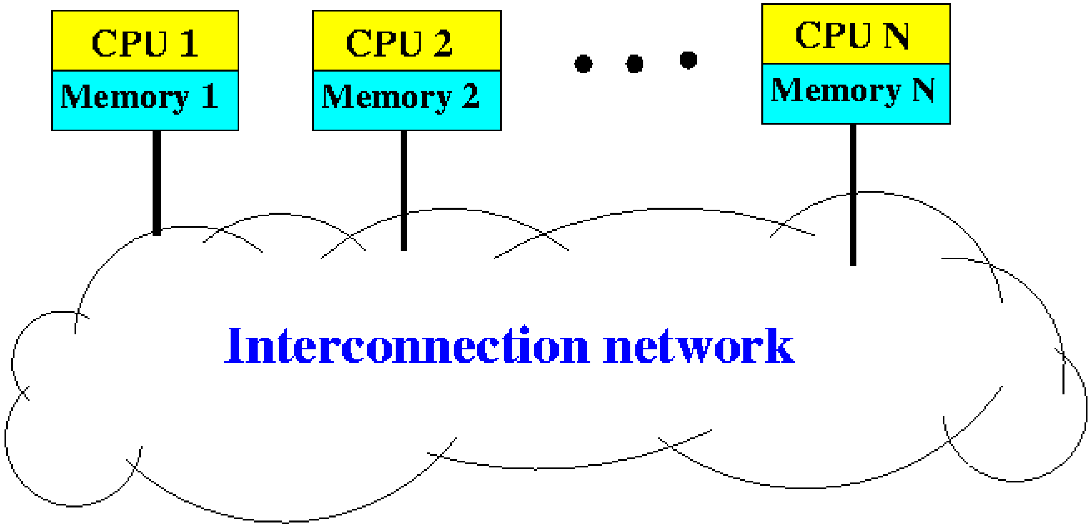
Execution Model

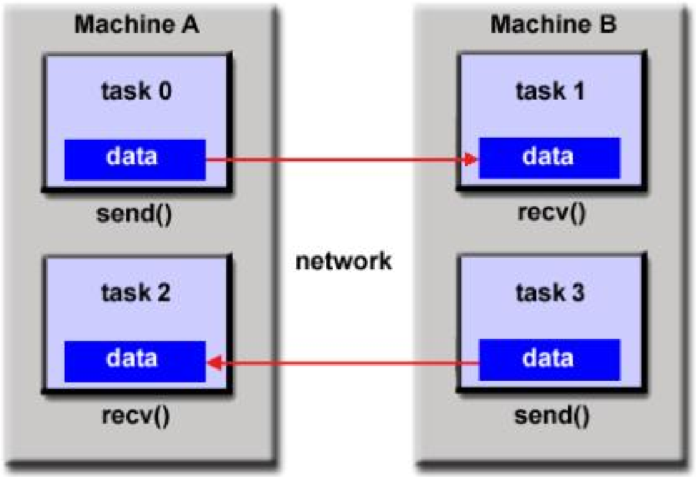


MIC

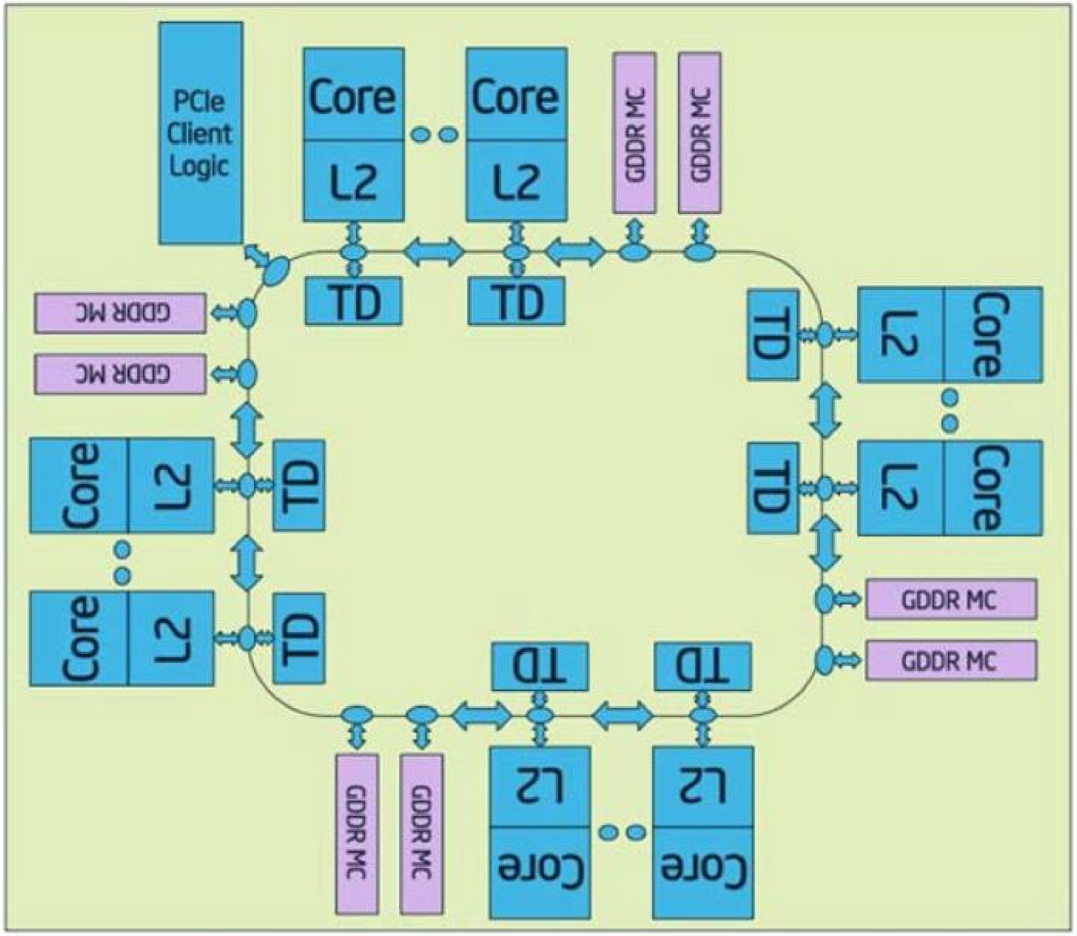


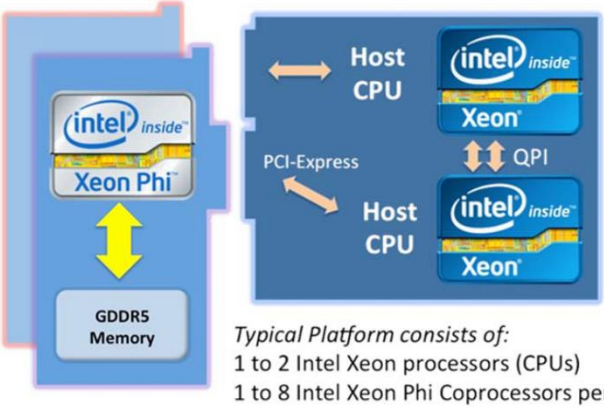
TASK Parallelism (MIMD)





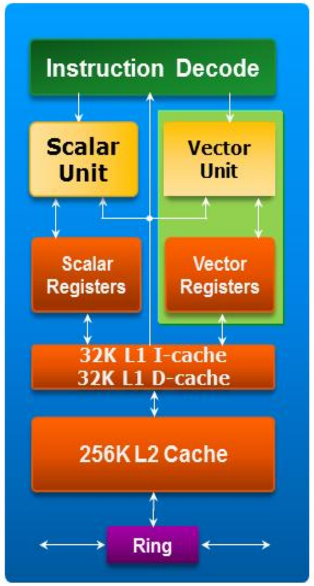
Xeon PHI Architecture



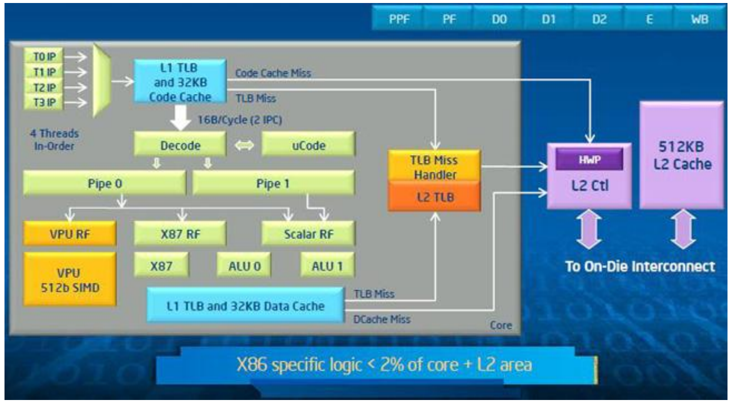


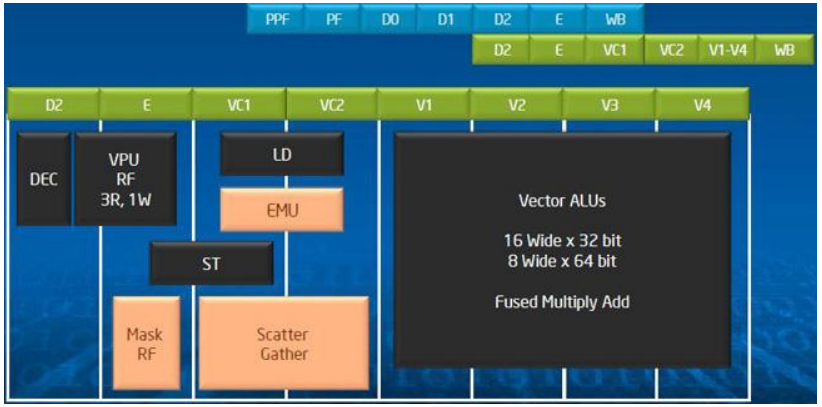
Typical Platform consists of:
1 to 2 Intel Xeon processors (CPUs)
1 to 8 Intel Xeon Phi Coprocessors per host

Core Architecture

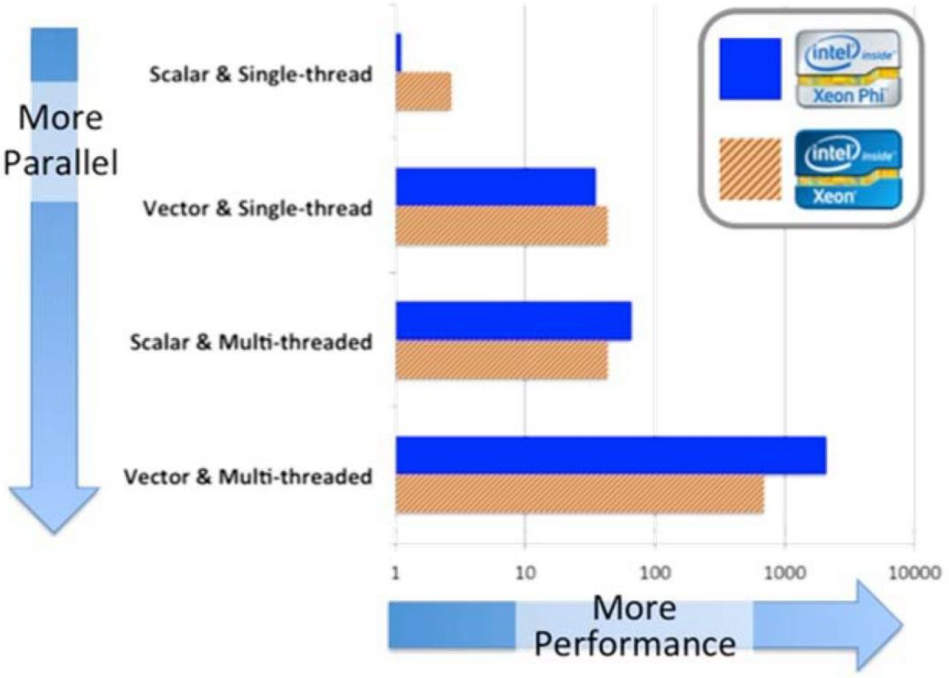


- Up to 32 in-order cores
- 4 hardware threads per core
- Two pipelines
  - Pentium® processor family-based scalar units
  - Fully-coherent L1 and L2 caches
  - 64-bit addressing
- All new vector unit
  - 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
  - 32 512-bit wide vector registers
    - Hold 16 singles or 8 doubles per register
  - Pipelined one-per-clock throughput
    - 4 clock latency, hidden by round-robin scheduling of threads
  - Dual issue with scalar instructions





A versatile combination



Execution Models

Offload Execution Mode

1. Host system offloads part or all of the computation from one or multiple processes or threads running on host
2. The application starts execution on the host
3. As the computation proceeds it can decide to send data to the coprocessor and let that work on it and the host and the coprocessor may or may not work in parallel.

OpenMP 4.0 TR being proposed and implemented in Intel® Composer XE provides directives to perform offload computations. Composer XE also provides some custom directives to perform offload operations.

Execution Models

Coprocessor Native Execution Mode

- An Xeon Phi hosts a Linux micro OS in it and can appear as another machine connected to the host like another node in a cluster.
- This execution environment allows the users to view the coprocessor as another compute node.
- In order to run natively, an application has to be cross compiled for Xeon Phi operating environment. Intel® Composer XE provides simple switch to generate cross compiled code.

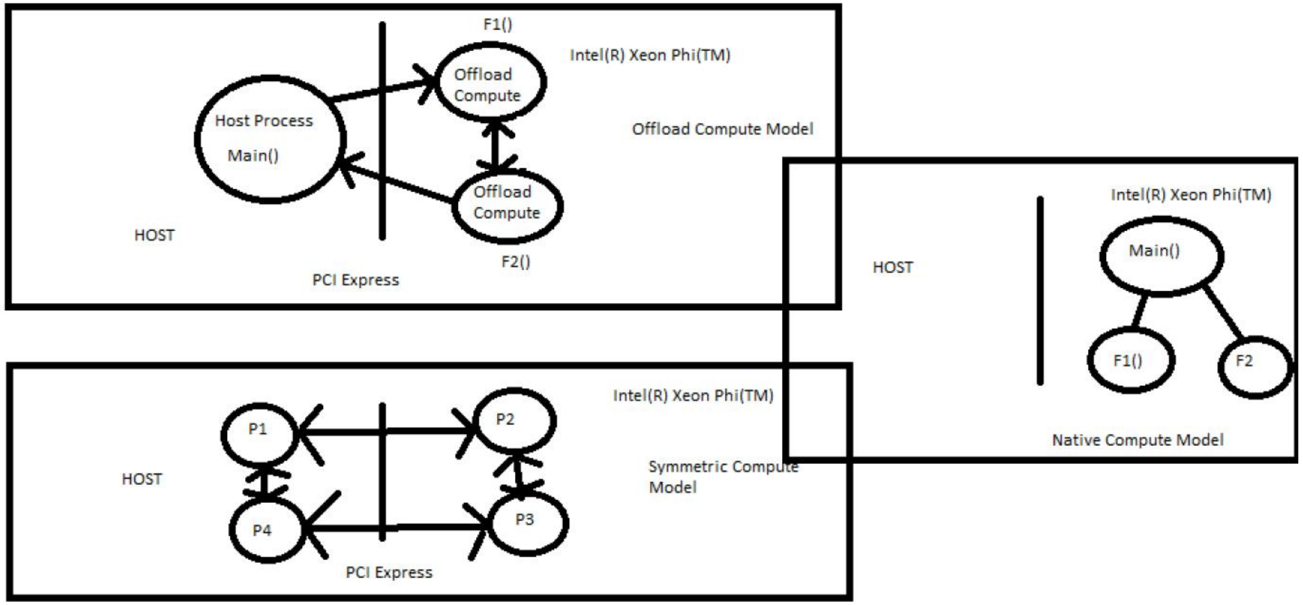
Execution Models

Symmetric Execution

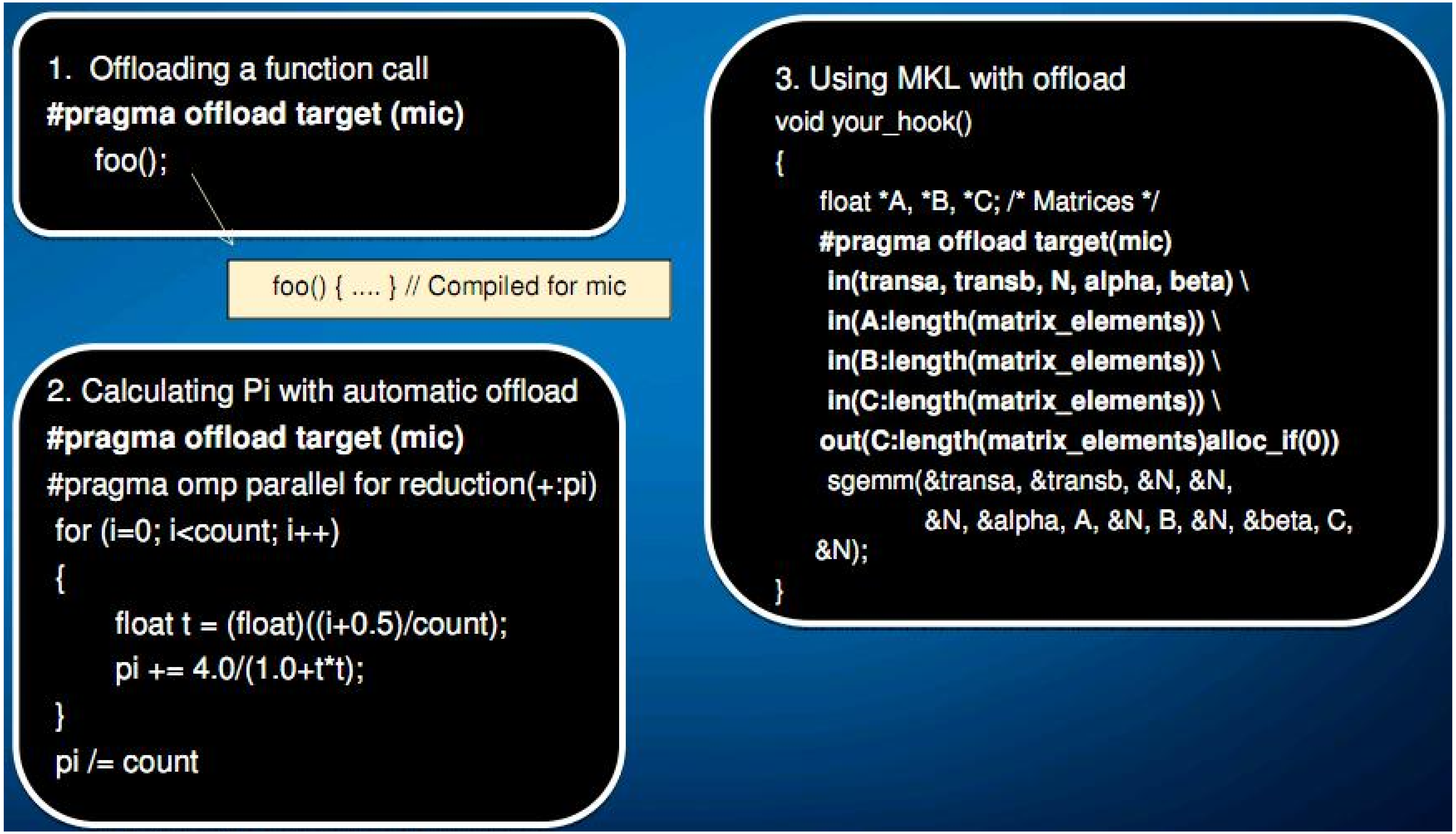
- The application processes run on both the host and the Phi coprocessor and communicate through some sort of message passing interface like MPI.
- This execution environment treats Xeon Phi card as another node in a cluster in a heterogeneous cluster environment.

Execution Models

Summary



Programming PHI



1. Offloading a function call

```
#pragma offload target (mic)
    foo();
```

```
foo() { .... } // Compiled for mic
```

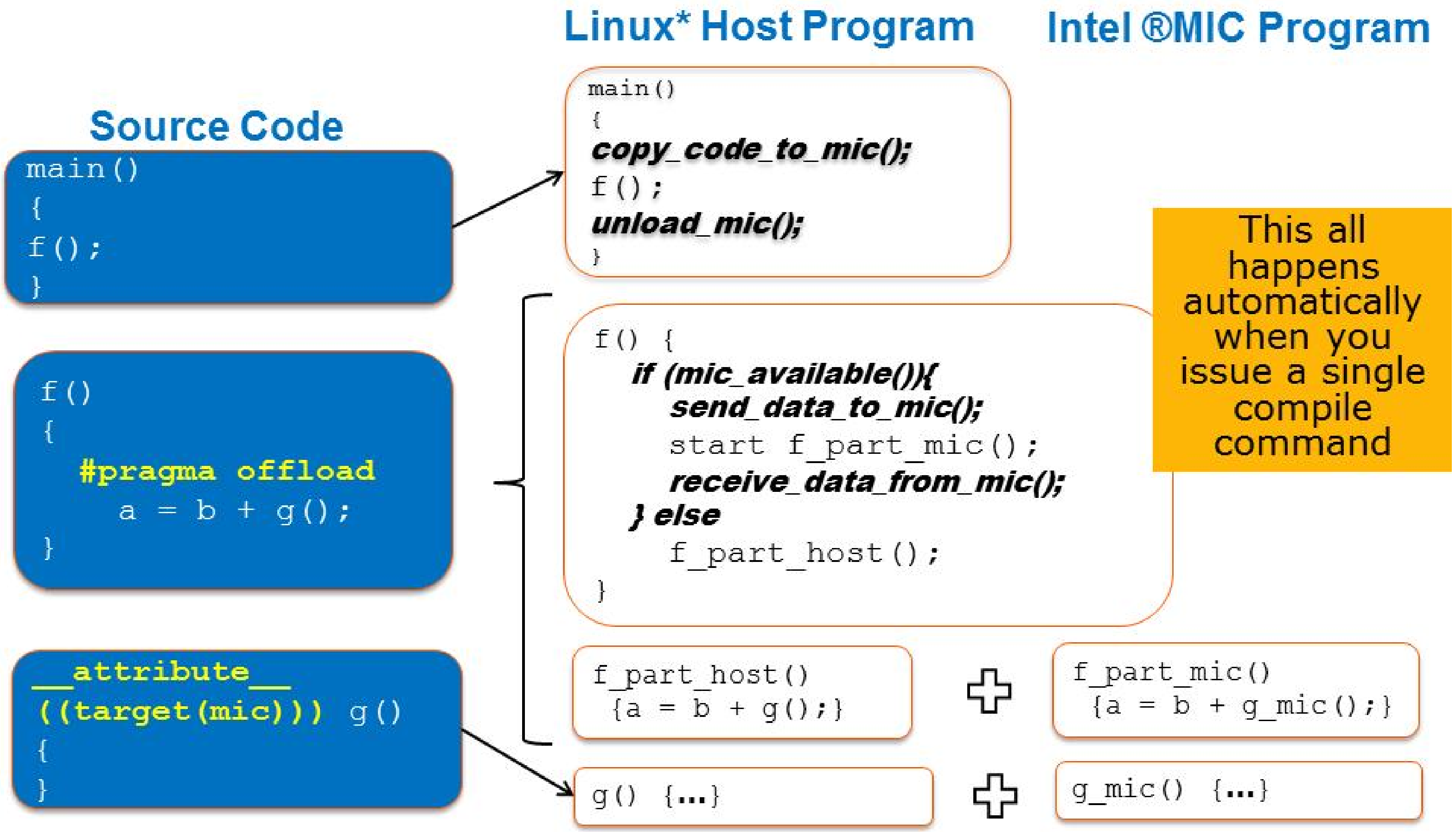
2. Calculating Pi with automatic offload

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
 for (i=0; i<count; i++)
 {
       float t = (float)((i+0.5)/count);
       pi += 4.0/(1.0+t*t);
 }
 pi /= count
```

3. Using MKL with offload

```
void your_hook()
{
     float *A, *B, *C; /* Matrices */
     #pragma offload target(mic)
      in(transa, transb, N, alpha, beta) \
      in(A:length(matrix_elements)) \
      in(B:length(matrix_elements)) \
      in(C:length(matrix_elements)) \
     out(C:length(matrix_elements)alloc_if(0))
      sgemm(&transa, &transb, &N, &N,
                 &N, &alpha, A, &N, B, &N, &beta, C,
     &N);
}
```

Heterogeneous Compiler



Linux* Host Program

Intel ®MIC Program

Source Code

```
main()
{
f();
}
```

```
f()
{
  #pragma offload
     a = b + g();
}
```

```
__attribute__
((target(mic))) g()
{
}
```

```
main()
{
copy_code_to_mic();
f();
unload_mic();
}
```

```
f() {
  if (mic_available()){
     send_data_to_mic();
     start f_part_mic();
     receive_data_from_mic();
  } else
     f_part_host();
}
```

```
f_part_host()
 {a = b + g();}
```

```
f_part_mic()
 {a = b + g_mic();}
```

```
g()  {…}
```

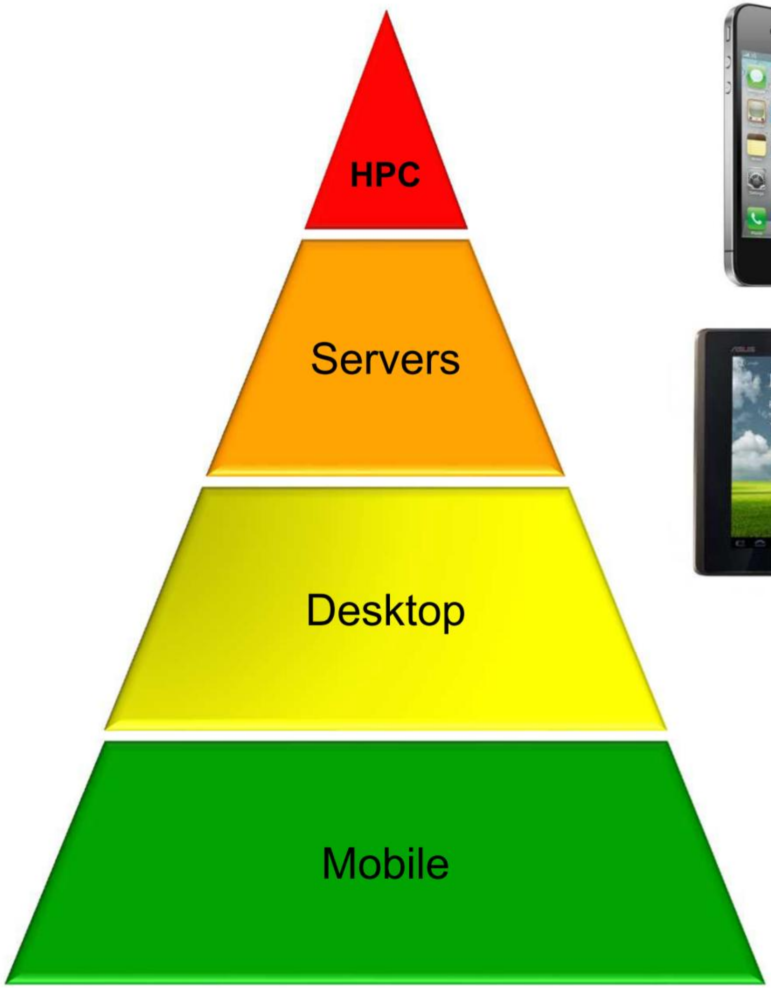
```
g_mic()  {…}
```

This all happens automatically when you issue a single compile command

ENERGY-EFFICIENT COMPUTING

ARM

Next Step in the Comodity Chain



HPC

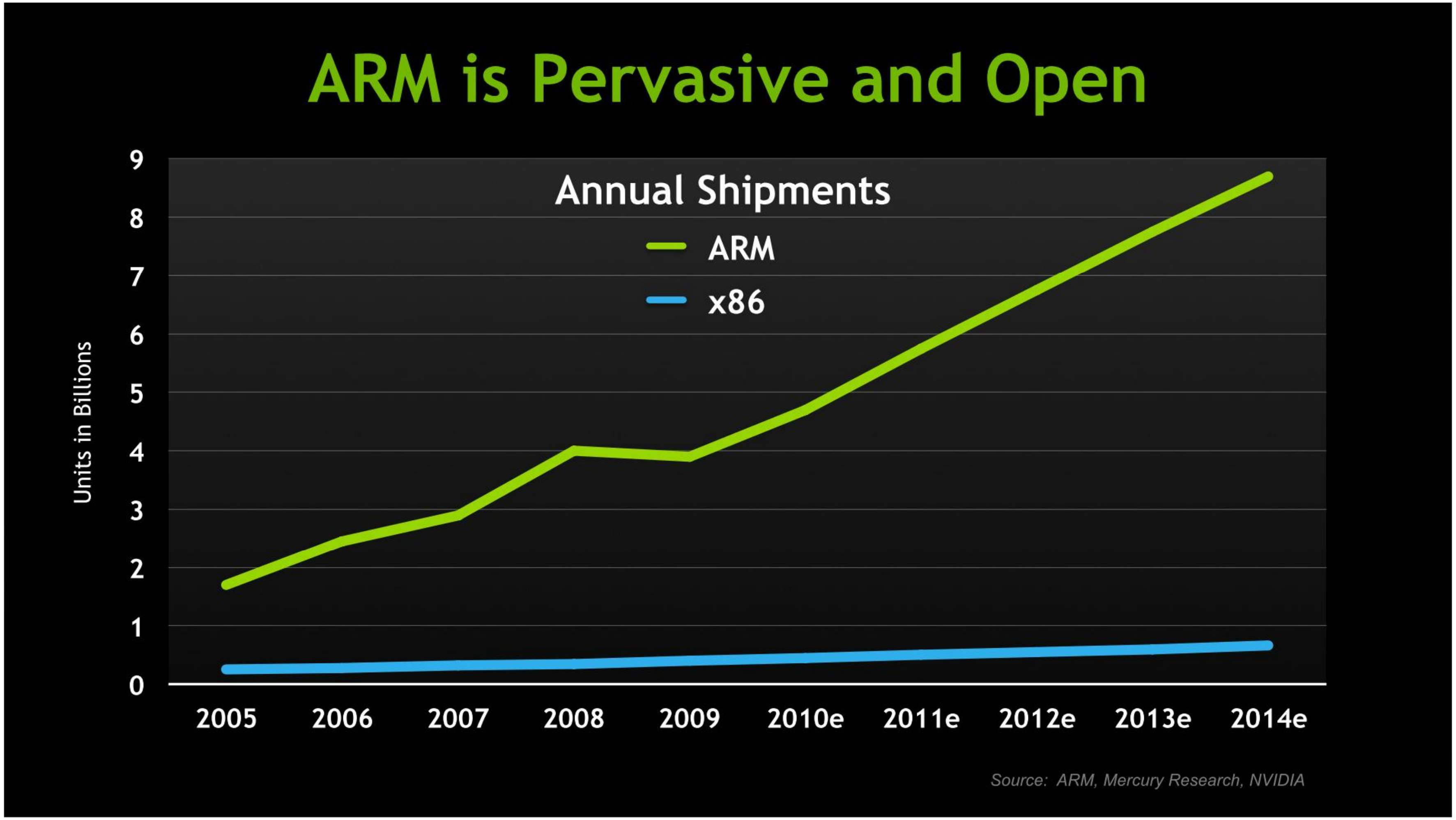
Servers

Desktop

Mobile

- Total cores in Jun'12 Top500
  - 13.5 Mcores
- Tablets sold in Q4 2011
  - 27 Mtablets
- Smartphones sold Q4 2011
  - > 100 Mphones

ARM is the most sold CPU Architecture



Source: ARM, Mercury Research, NVIDIA

ARM vs x86

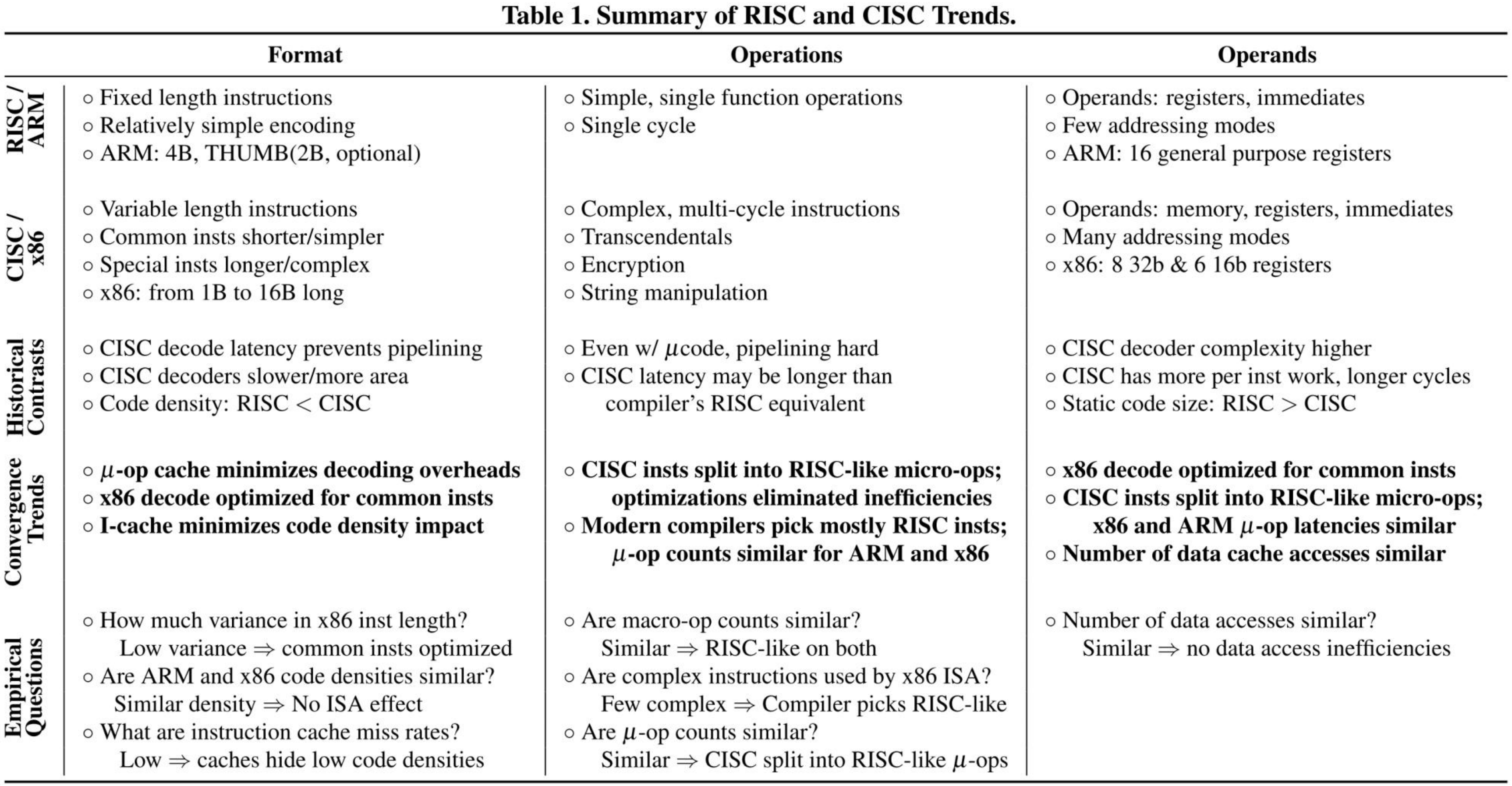
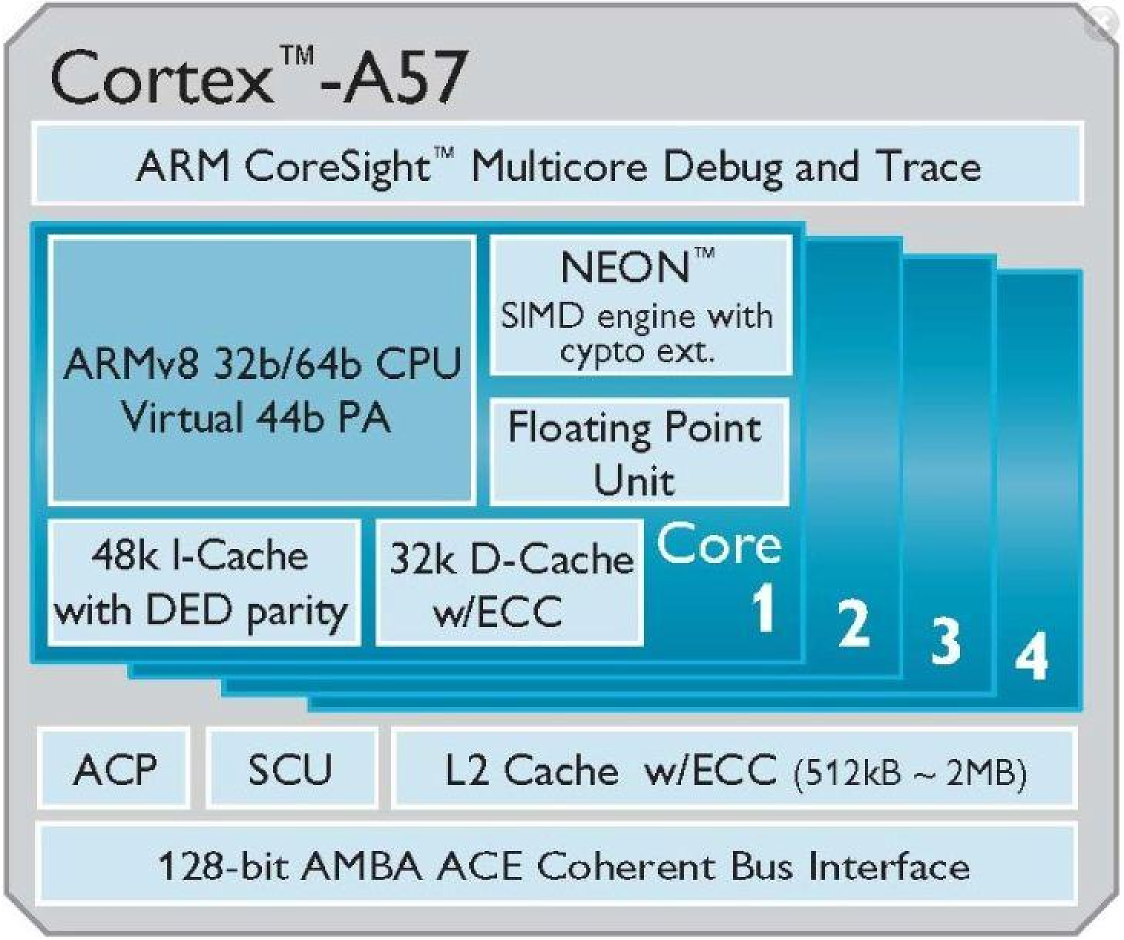
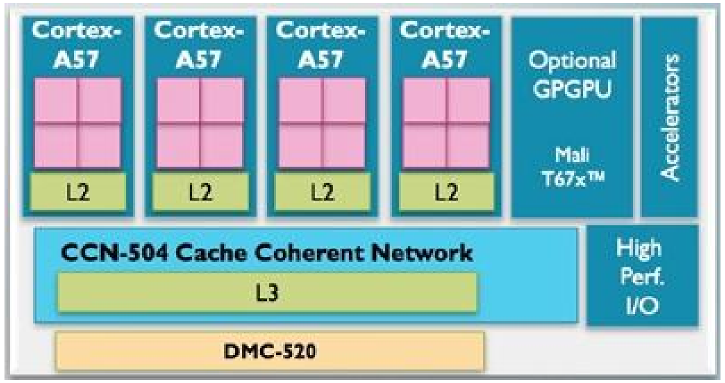


Table 1. Summary of RISC and CISC Trends.

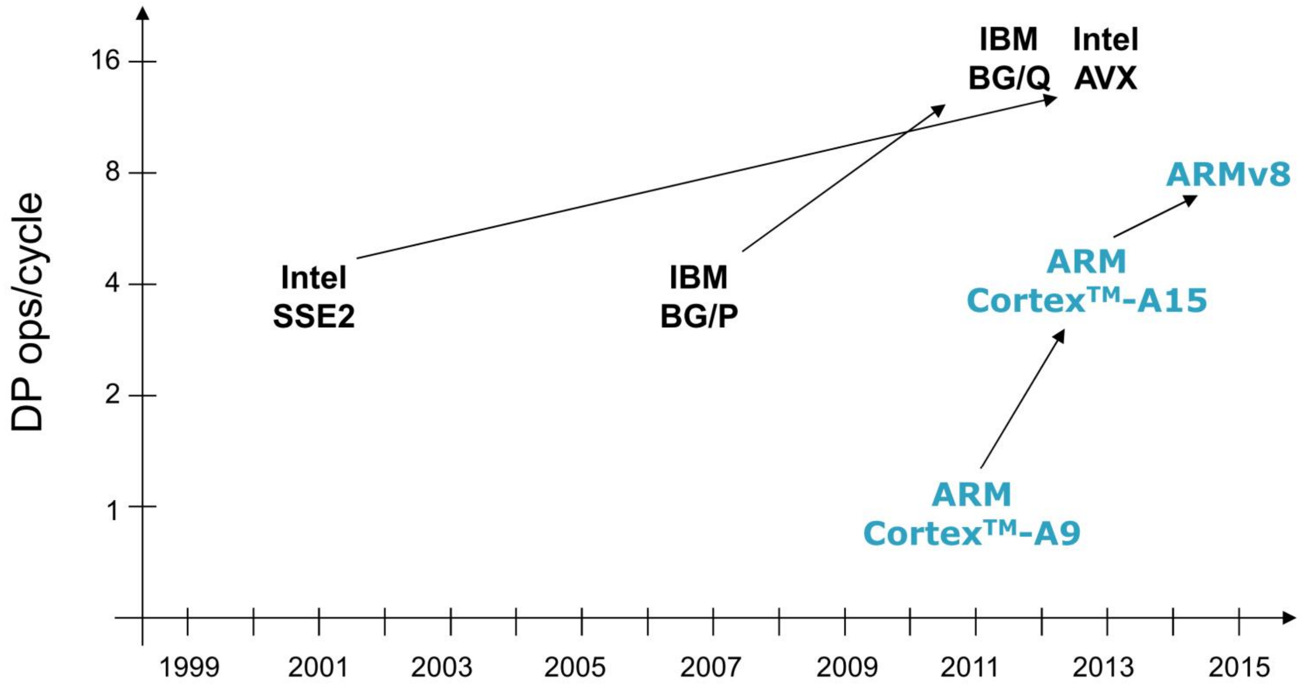
	Format	Operations	Operands
RISC / ARM	○ Fixed length instructions ○ Relatively simple encoding ○ ARM: 4B, THUMB(2B, optional)	○ Simple, single function operations ○ Single cycle	○ Operands: registers, immediates ○ Few addressing modes ○ ARM: 16 general purpose registers
CISC / x86	○ Variable length instructions ○ Common insts shorter/simpler ○ Special insts longer/complex ○ x86: from 1B to 16B long	○ Complex, multi-cycle instructions ○ Transcendentals ○ Encryption ○ String manipulation	○ Operands: memory, registers, immediates ○ Many addressing modes ○ x86: 8 32b & 6 16b registers
Historical Contrasts	○ CISC decode latency prevents pipelining ○ CISC decoders slower/more area ○ Code density: RISC < CISC	○ Even w/ μcode, pipelining hard ○ CISC latency may be longer than compiler's RISC equivalent	○ CISC decoder complexity higher ○ CISC has more per inst work, longer cycles ○ Static code size: RISC > CISC
Convergence Trends	○ **μ-op cache minimizes decoding overheads** ○ **x86 decode optimized for common insts** ○ **I-cache minimizes code density impact**	○ **CISC insts split into RISC-like micro-ops; optimizations eliminated inefficiencies** ○ **Modern compilers pick mostly RISC insts; μ-op counts similar for ARM and x86**	○ **x86 decode optimized for common insts** ○ **CISC insts split into RISC-like micro-ops; x86 and ARM μ-op latencies similar** ○ **Number of data cache accesses similar**
Empirical Questions	○ How much variance in x86 inst length? Low variance ⇒ common insts optimized ○ Are ARM and x86 code densities similar? Similar density ⇒ No ISA effect ○ What are instruction cache miss rates? Low ⇒ caches hide low code densities	○ Are macro-op counts similar? Similar ⇒ RISC-like on both ○ Are complex instructions used by x86 ISA? Few complex ⇒ Compiler picks RISC-like ○ Are μ-op counts similar? Similar ⇒ CISC split into RISC-like μ-ops	○ Number of data accesses similar? Similar ⇒ no data access inefficiencies

High Performance ARM CPU



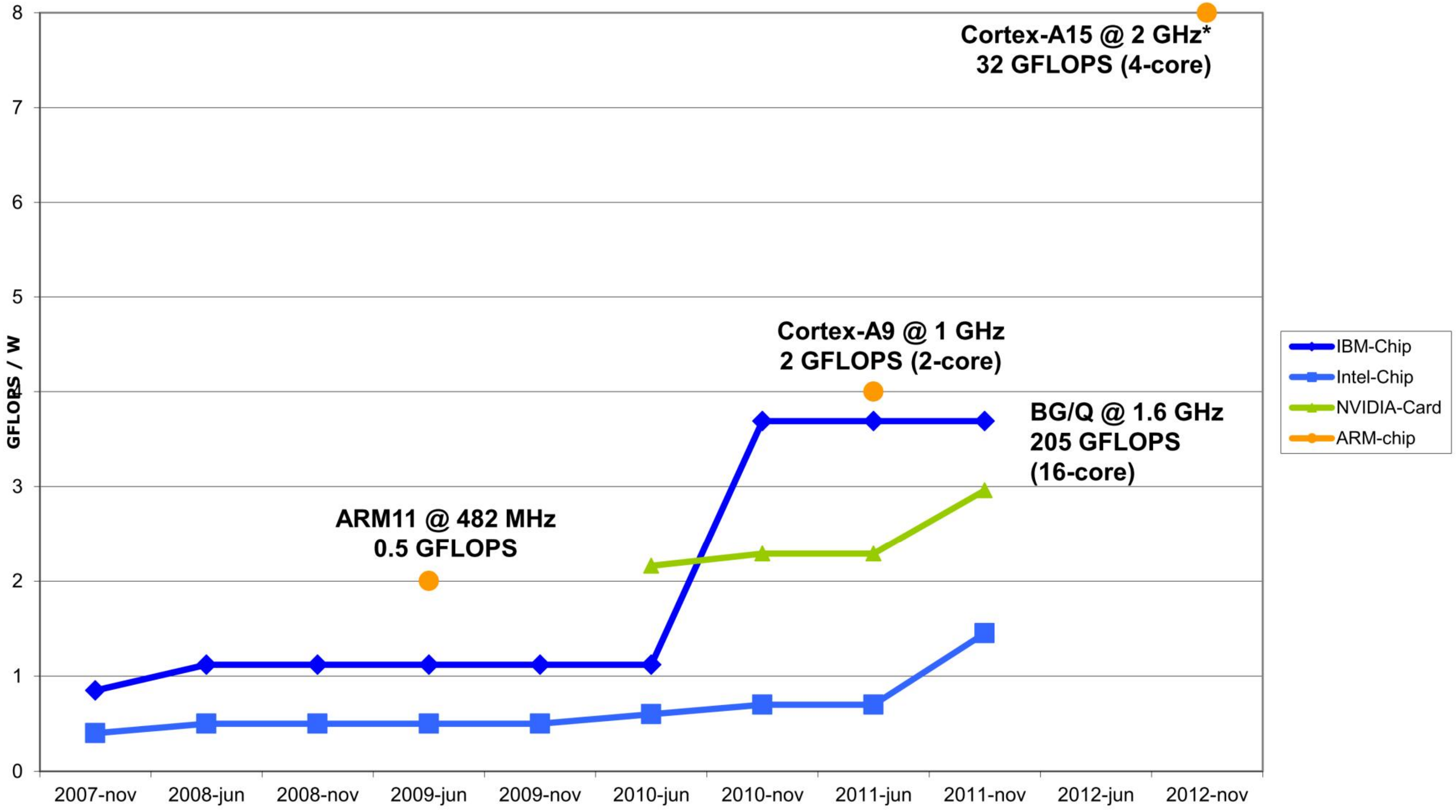


ARM Improvement in DFP



- IBM BG/Q and Intel AVX implement DP in 256-bit SIMD
  - 8 DP ops / cycle
- ARM quickly moved from optional floating-point to state-of-the-art
  - ARMv8 ISA introduces DP in the NEON instruction set (128-bit SIMD)

ARM Energy Efficiency



Cortex-A15 @ 2 GHz*
32 GFLOPS (4-core)

Cortex-A9 @ 1 GHz
2 GFLOPS (2-core)

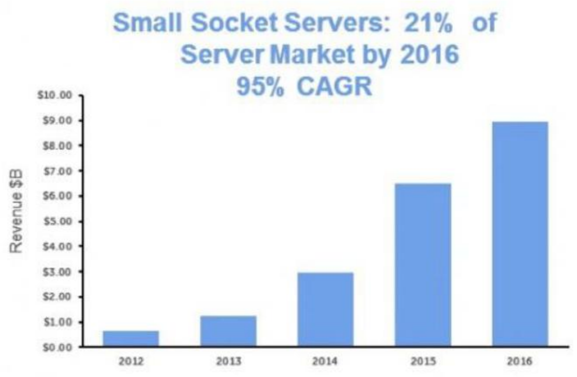
BG/Q @ 1.6 GHz
205 GFLOPS
(16-core)

ARM11 @ 482 MHz
0.5 GFLOPS

SM - Summary

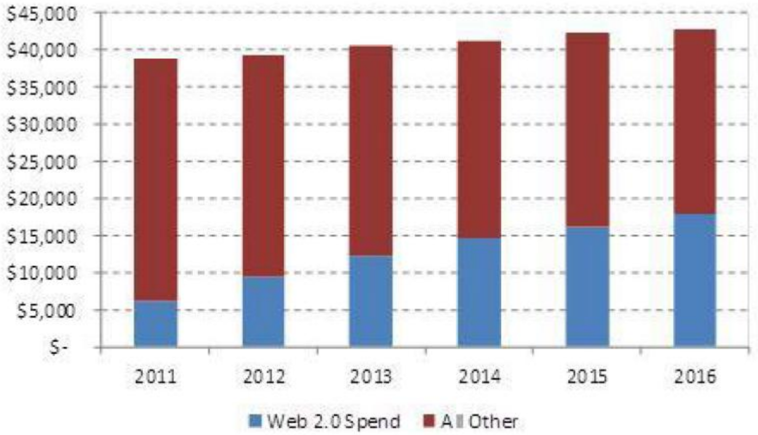
Vendor	2011 Revenue	Market Share	2010 Revenue	Market Share	2011/2010 Revenue Growth
1. IBM	$16,456	31.5%	$15,342	31.1%	7.3%
2. HP	$15,301	29.3%	$15,388	31.2%	-0.6%
3. Dell	$7,814	15.0%	$7,106	14.4%	10.0%
4. Oracle	$3,215	6.2%	$3,204	6.5%	0.3%
5. Fujitsu	$2,516	4.8%	$2,197	4.4%	14.5%
Others	$6,964	13.3%	$6,159	12.5%	13.1%

According to IDC, there were 8.05M x86 servers shipped in 2011. At a blended average price of $4,837, the x86 server TAM was $39B.



Small Socket Servers: 21% of Server Market by 2016 95% CAGR

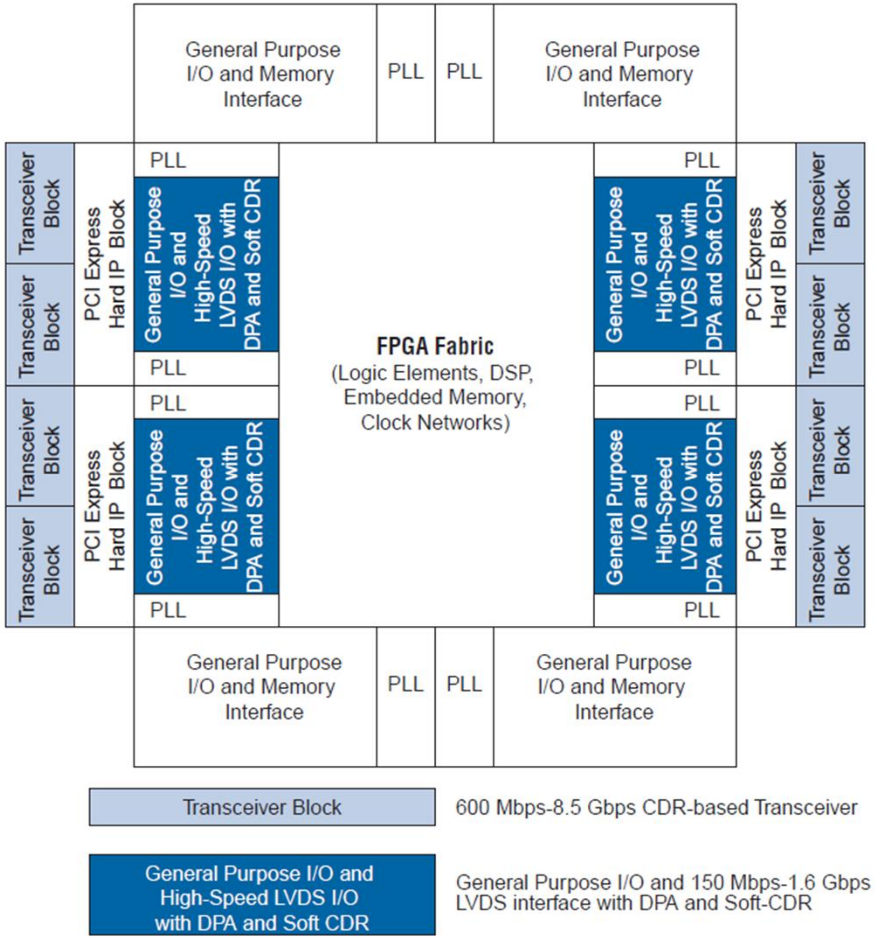
Source: Oppenheimer, Cloudy with a Chance of ARM, March 30th 2012



FPGAs

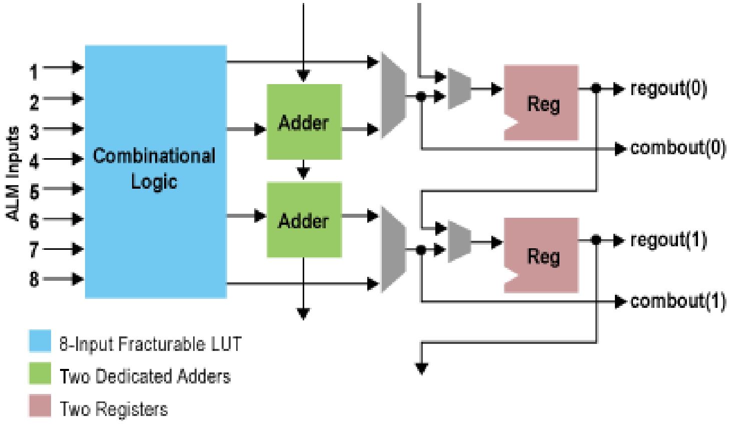
Altera Stratix IV

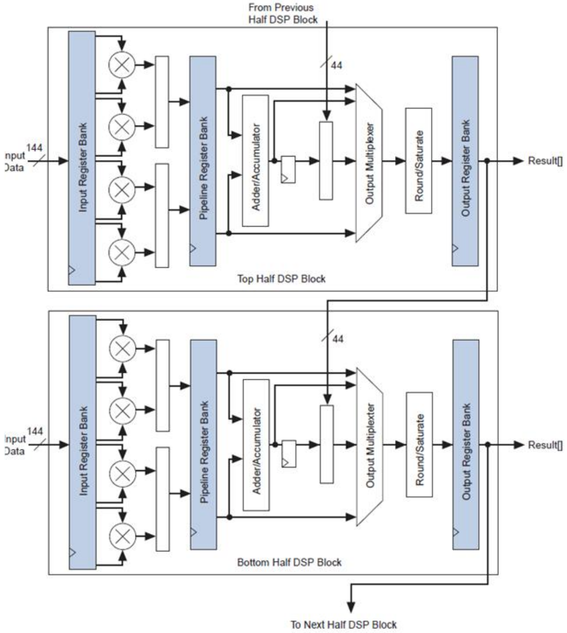
Altera Stratix IV GX 230 / 1152 pins


General Purpose I/O and Memory Interface
PLL
PLL
General Purpose I/O and Memory Interface
Transceiver Block
Transceiver Block
PCI Express Hard IP Block
PLL
General Purpose I/O and High-Speed LVDS I/O with DPA and Soft CDR
PLL
FPGA Fabric
(Logic Elements, DSP, Embedded Memory, Clock Networks)
Transceiver Block
Transceiver Block
PCI Express Hard IP Block
Transceiver Block
Transceiver Block
General Purpose I/O and Memory Interface
Transceiver Block
600 Mbps-8.5 Gbps CDR-based Transceiver
General Purpose I/O and High-Speed LVDS I/O with DPA and Soft CDR
General Purpose I/O and 150 Mbps-1.6 Gbps LVDS interface with DPA and Soft-CDR

Stratix IV FPGA main features

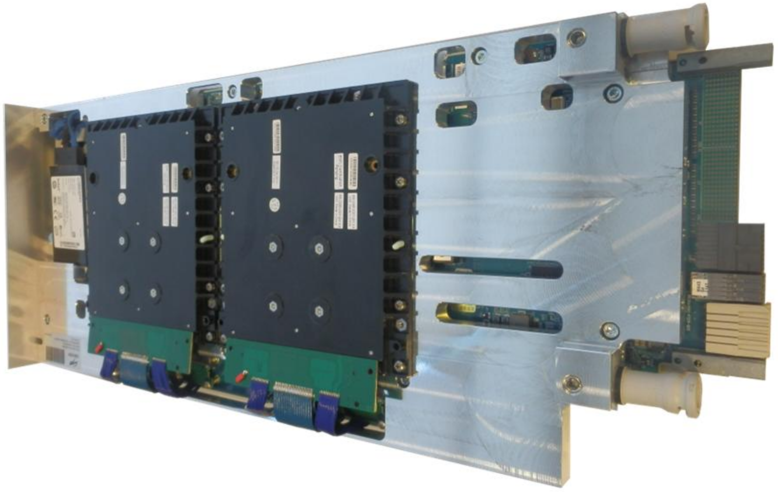
- 91200 ALMs (Adaptive Logic Modules)
- 18 MB SRAM
- 228000 Logic Elements (~ gates)
- High speed transceivers
- 2 PCIe 2.0 hard IP blocks
- 161 DSP blocks,
  (322 36*36 mult 1288 18*18 mult. etc
  up to 55GFlops Matrix Inversion)
  All of them can be used in parallel





The Aurora Tigon node card

Node card
2 X Intel Xeon E5 series
2 X Nvidia Kepler K20 or
2 X Intel Xeon Phi 5120D
2730 GFlops
800 W power consumption
64 GB soldered memory

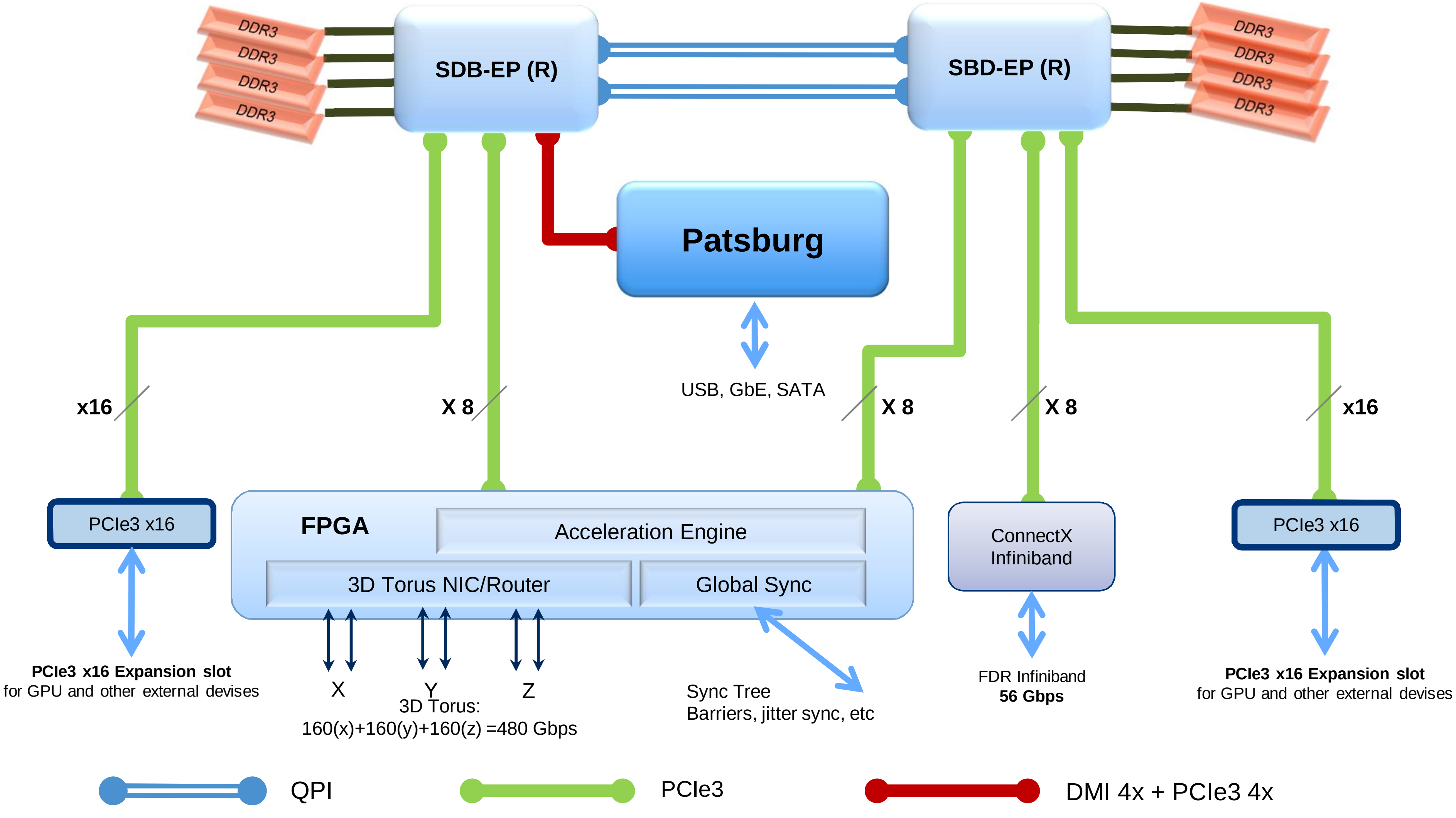


- The **node card** is the main processing unit of an Eurotech Aurora Tigon machine

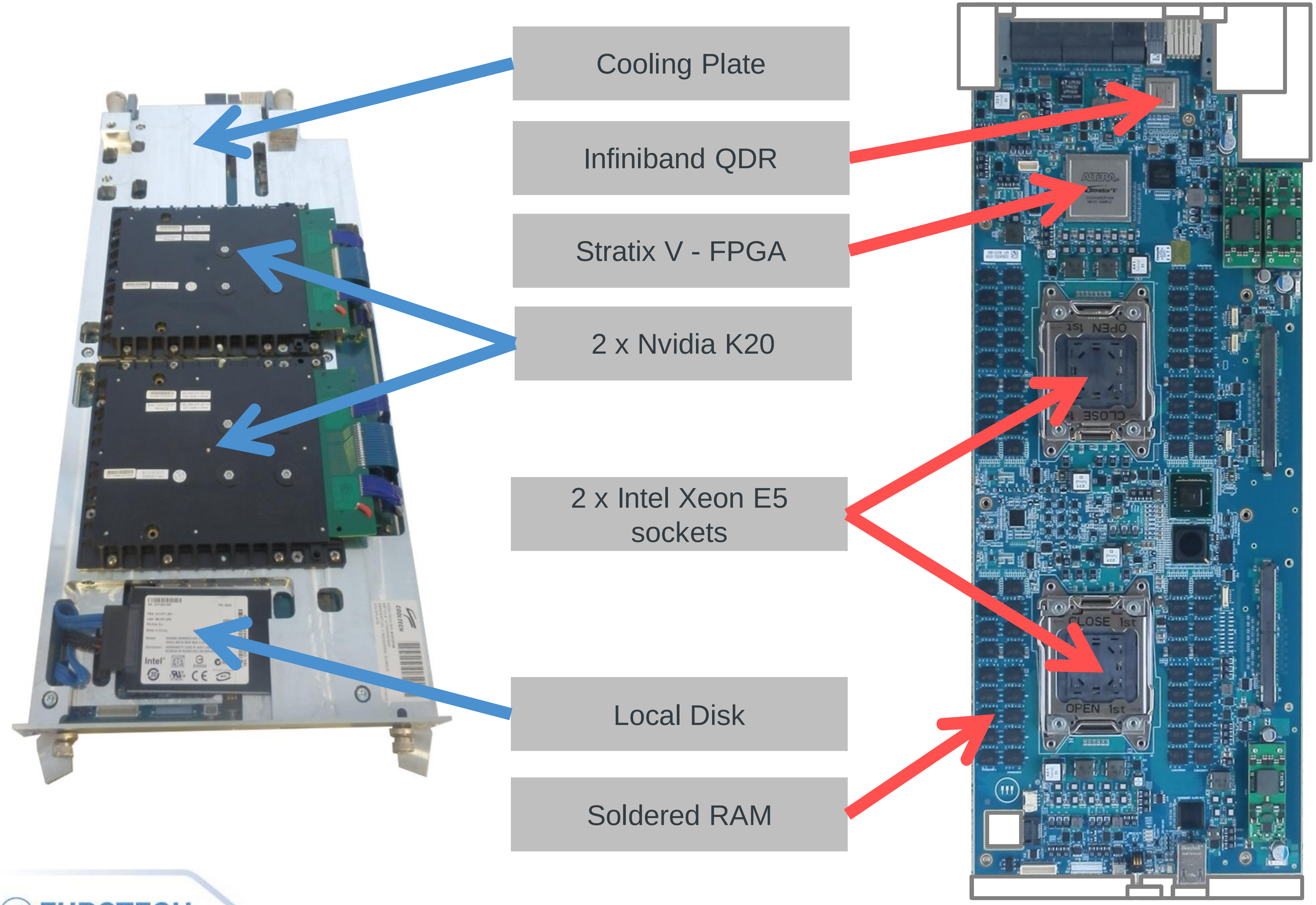
- An aluminum cold plate that cools the board smoothing temperature distributions and assuring maximum heat extraction efficacy

- A large, high end FPGA allow implementation of a point to point 3D-Torus network

Aurora Tigon – Node Card Architecture



The Aurora Tigon node card



Cooling Plate

Infiniband QDR

Stratix V - FPGA

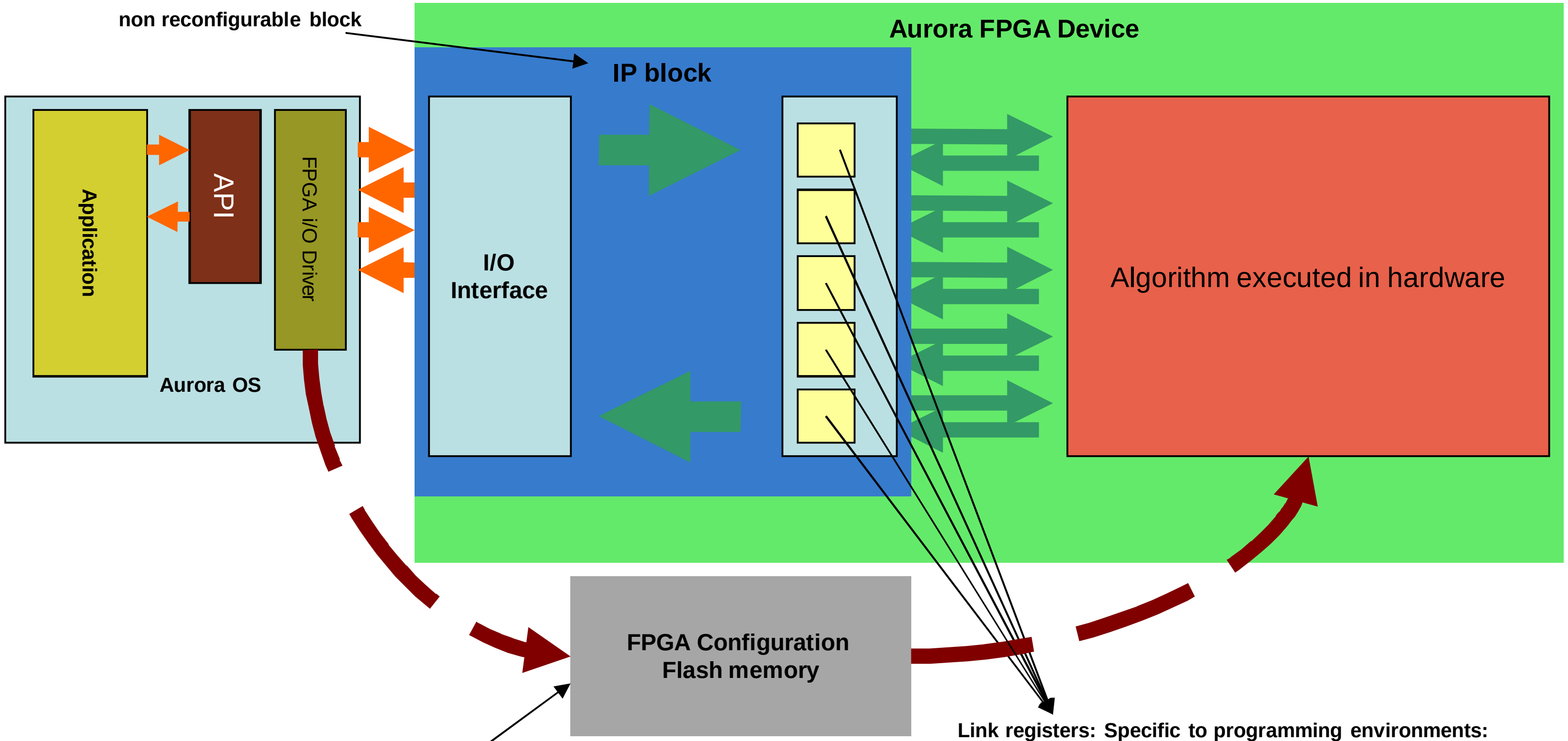
2 x Nvidia K20

2 x Intel Xeon E5 sockets

Local Disk

Soldered RAM

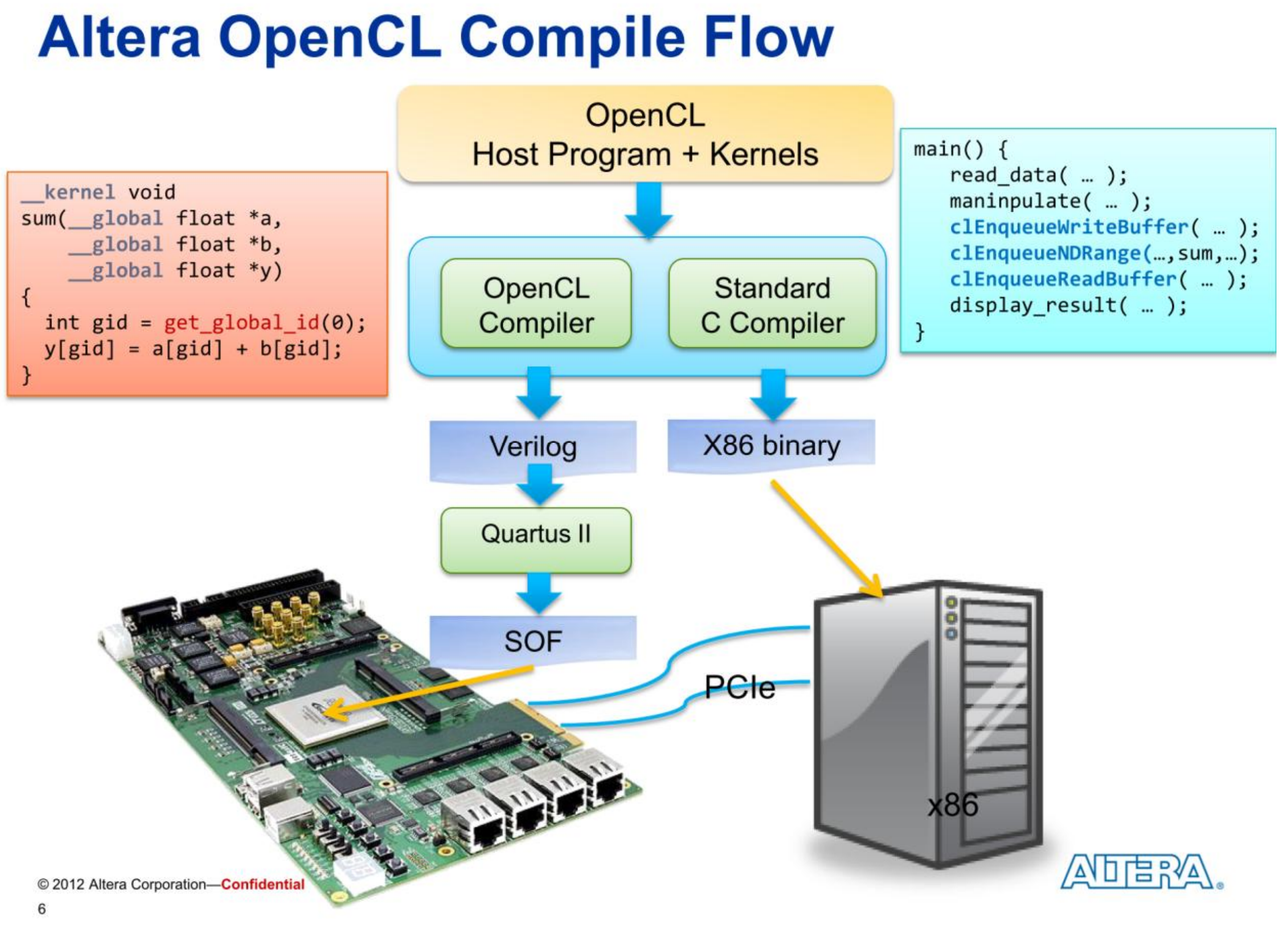
Aurora Accelerator core



SW FPGA Configuration using Aurora node PCIe interfaces

Link registers: Specific to programming environments: key to easy integration with C code to VHDL (hardware instantiation) programming interface

OpenCL and FPGAs

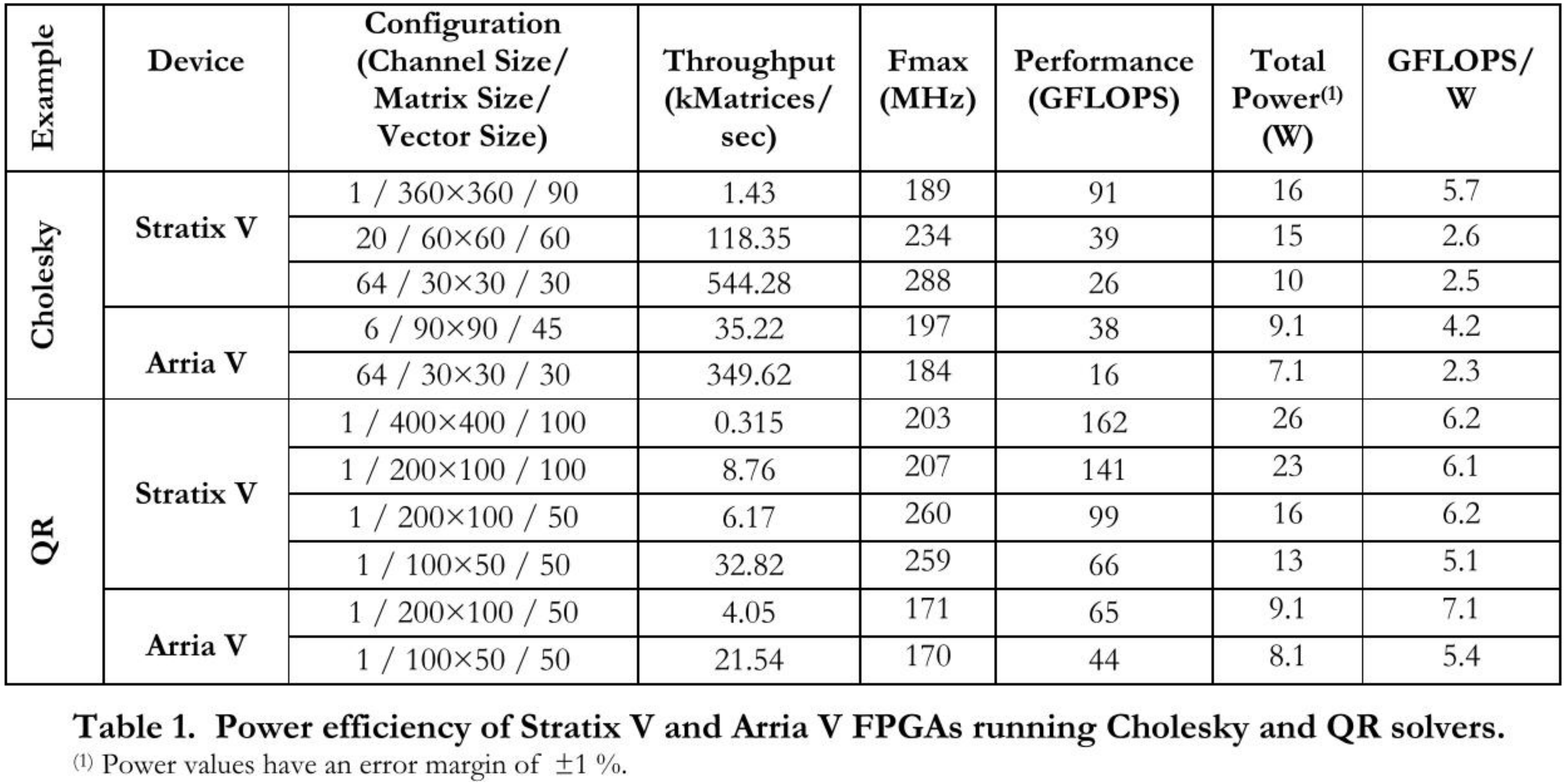


Altera OpenCL Compile Flow

```
__kernel void
sum(__global float *a,
    __global float *b,
    __global float *y)
{
  int gid = get_global_id(0);
  y[gid] = a[gid] + b[gid];
}
```

```
main() {
   read_data( … );
   maninpulate( … );
   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…,sum,…);
   clEnqueueReadBuffer( … );
   display_result( … );
}
```

Energy Efficient FP Computing



Example	Device	Configuration (Channel Size/ Matrix Size/ Vector Size)	Throughput (kMatrices/ sec)	Fmax (MHz)	Performance (GFLOPS)	Total Power[(1)] (W)	GFLOPS/ W
Cholesky	Stratix V	1 / 360×360 / 90	1.43	189	91	16	5.7
		20 / 60×60 / 60	118.35	234	39	15	2.6
		64 / 30×30 / 30	544.28	288	26	10	2.5
	Arria V	6 / 90×90 / 45	35.22	197	38	9.1	4.2
		64 / 30×30 / 30	349.62	184	16	7.1	2.3
QR	Stratix V	1 / 400×400 / 100	0.315	203	162	26	6.2
		1 / 200×100 / 100	8.76	207	141	23	6.1
		1 / 200×100 / 50	6.17	260	99	16	6.2
		1 / 100×50 / 50	32.82	259	66	13	5.1
	Arria V	1 / 200×100 / 50	4.05	171	65	9.1	7.1
		1 / 100×50 / 50	21.54	170	44	8.1	5.4

Table 1. Power efficiency of Stratix V and Arria V FPGAs running Cholesky and QR solvers.

[(1)] Power values have an error margin of ±1 %.

Thank You

www.eurotech.com