GPU for Scientific Computing

Alessandro Lonardo, INFN Roma 1

ESC 2011
Bertinoro, 24 October 2011

Outline

- What is GPGPU computing
- Motivation behind GPUs as general purpose computing engines
- HPC results
- How did we get here? Some History
- Current GPU architectures
- CUDA C example.
- Libraries.
- INFN projects using GPU technology.

What is GPGPU Computing?
Some definitions

GPGPU

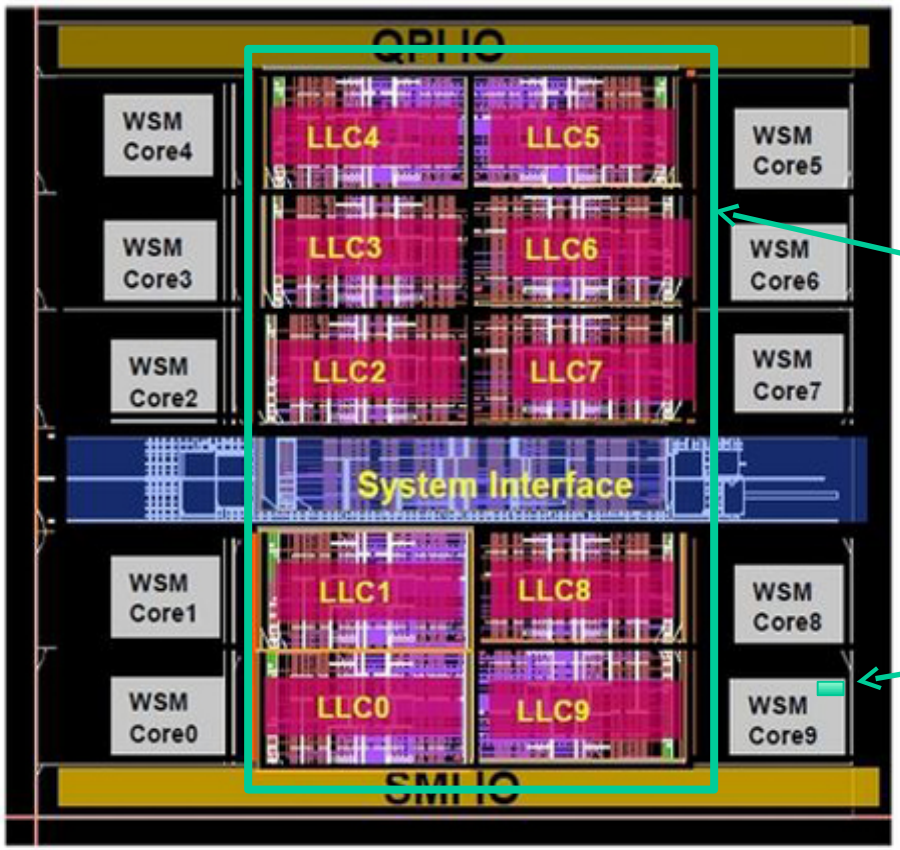
From Wikipedia, the free encyclopedia

General-purpose computing on graphics processing units (**GPGPU**, also referred to as **GPGP** and less often **GP²U**) is the technique of using a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU. It is made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines, which allows programmers to use stream processing on non-graphics data[1][2][3]. Additionally, the use of multiple graphics cards in a single computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing[4]

- *Stream Processing*:
  - *Stream*: set of data.
  - *Kernel Functions*: a series of operations applied to each element of the stream.

Why GPUs are interesting Computing Engines? Today's CPUs

- ~2003: the 'free lunch' of CPU clock scaling is over.
- CPU architectures moved towards multi-core.
- Today you can buy: 10 cores Intel Wesmere-Ex

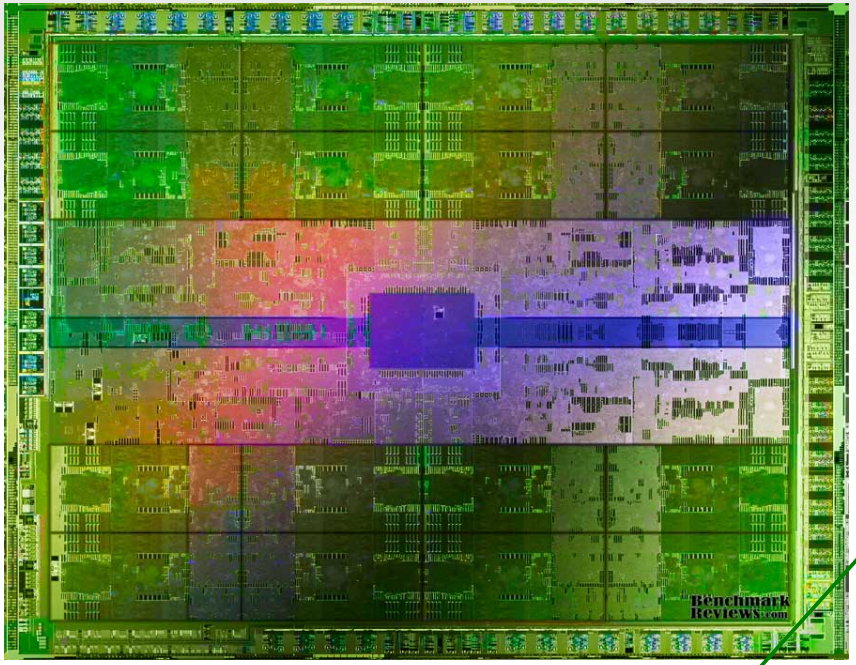


Lots of Caches

Few processing: 4 FP units are probably 1 pixel wide

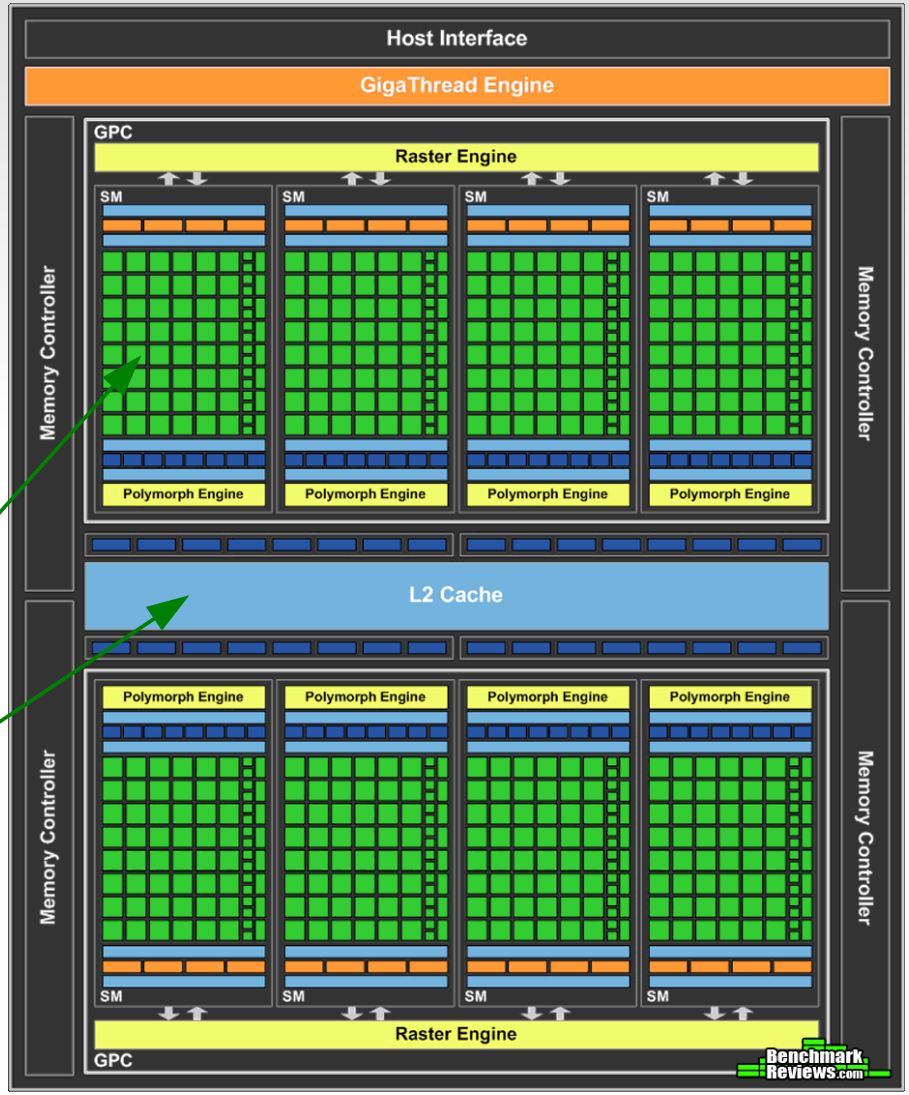
Why GPUs are interesting Computing Engines?
Today's GPUs

Nvidia Fermi GF100



Lots of computing units

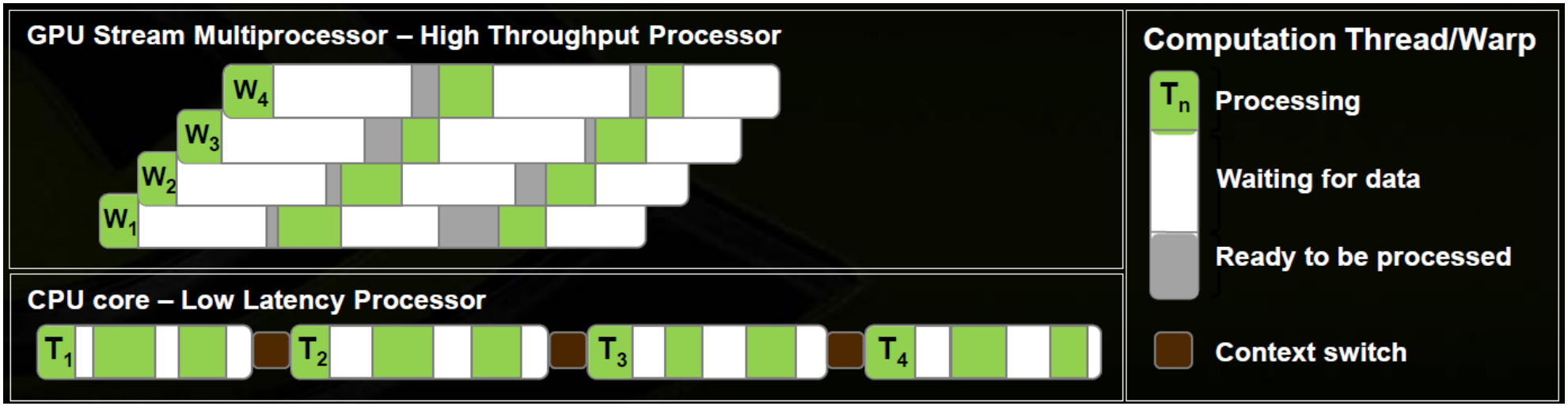
Small Cache



Why GPUs are interesting Computing Engines? CPU vs. GPUs

- O(10) cores
- Low latency access to cached data sets
- Out of order and speculative execution control logic
- Good for *task parallelism*

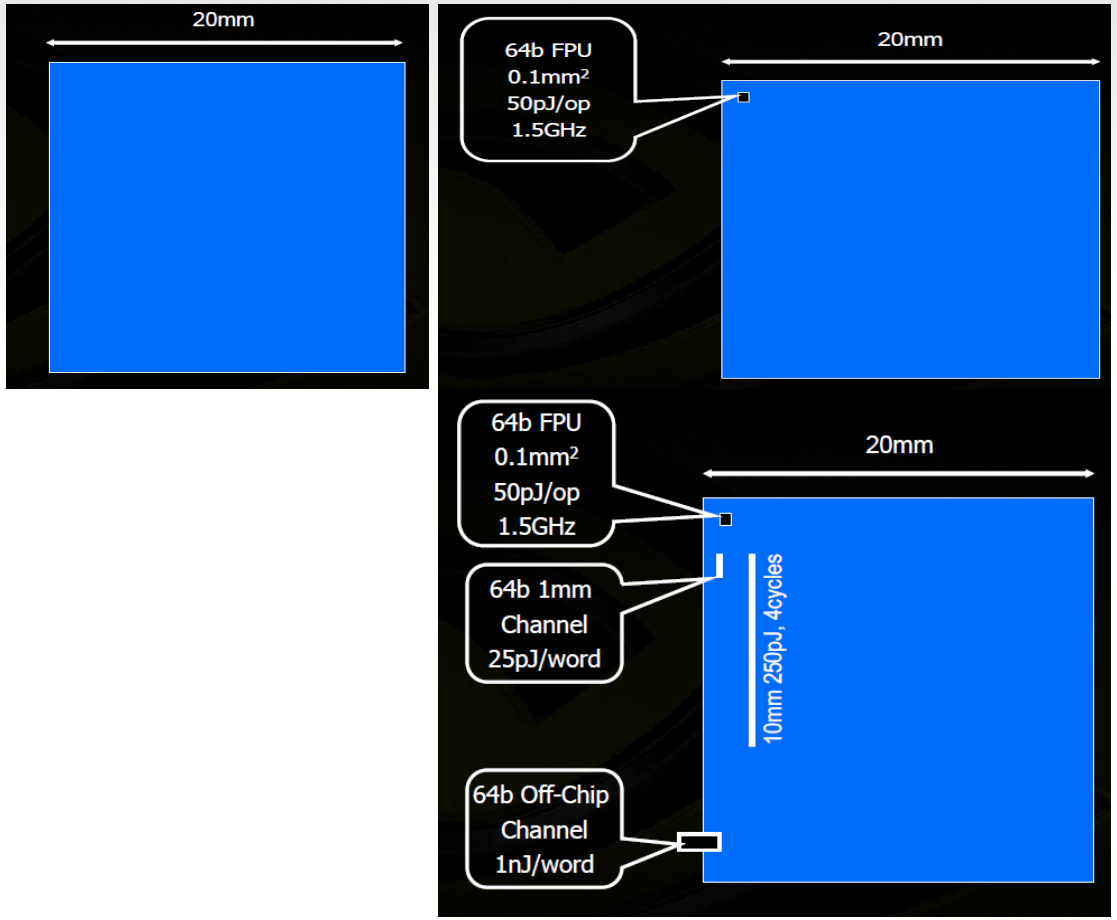
- O(100) cores
- Fast on-board memories
- Architecture allows hiding memory latencies
- Good for *data parallelism*



Why GPUs are interesting Computing Engines? Some theory

*"chips are power limited and most power is spent moving data around"**

- 4 cm2 chip
- 4000 64bit FPU fit
- Moving 64bits on chip == 10FMAs
- Moving 64bits off chip == 20FMAs



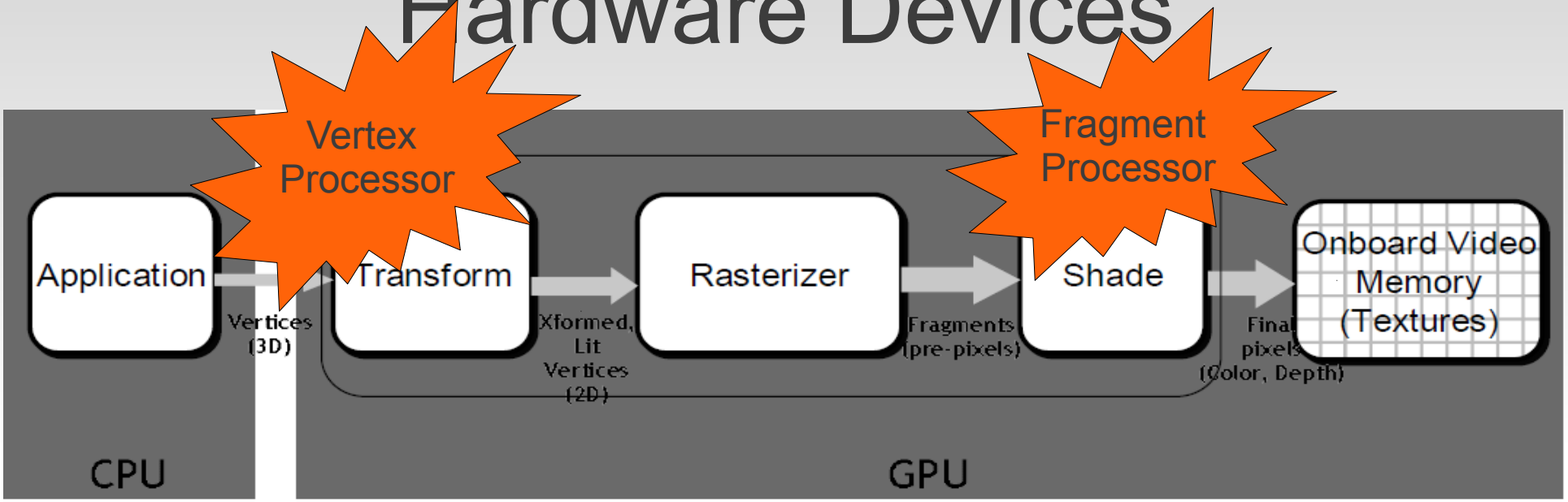
*Bill Dally, Nvidia Corp. talk at SC09

Why GPUs are interesting Computing Engines? Check the results!

	NAME/MANUFACTURER/COMPUTER	LOCATION	COUNTRY	CORES	R_{max} Pflop/s
1	**K Computer** SPARC64 VIIIfx 2.0GHz, Tofu interconnect	RIKEN	Japan	548,352	8.16
2	**Tianhe-1A** 6-core Intel X5670 2.93 GHz + Nvidia M2050 GPU w/custom interconnect	NUDT/NSCC/Tianjin	China	186,368	2.56
3	**Jaguar** Cray XT-5 6-core AMD 2.6 GHz w/custom interconnect	DOE/SC/ORNL	USA	224,162	1.76
4	**Nebulae** Dawning TC3600 Blade Intel X5650 2.67 GHz, NVidia Tesla C2050 GPU w/ Iband	NSCS	China	120,640	1.27
5	**Tsubame 2.0** HP Proliant SL390s G7 nodes (Xeon X5670 2.93GHz) , NVIDIA Tesla M2050 GPU w/Iband	TiTech	Japan	73,278	1.19

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	2097.19	IBM Thomas J. Watson Research Center	NNSA/SC Blue Gene/Q Prototype 2	40.95
2	1684.20	IBM Thomas J. Watson Research Center	NNSA/SC Blue Gene/Q Prototype 1	38.80
3	1375.88	Nagasaki University	DEGIMA Cluster, Intel i5, ATI Radeon GPU, Infiniband QDR	34.24
4	958.35	GSIC Center, Tokyo Institute of Technology	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows	1243.80
5	891.88	CINECA / SCS - SuperComputing Solution	iDataPlex DX360M3, Xeon 2.4, nVidia GPU, Infiniband	160.00

History
Hardware Devices



- Fixed Rendering Pipeline (1999-2000): Nvidia (NV10) GeForce 256, ATI (R100) .
- Programmable Rendering Pipeline (2001): Nvidia (NV20) GeForce 3, ATI (R200) Radeon 8500.
- Floating Point Performance-Cg programming (2002): Nvidia (NV30) GeForce Fx, ATI (R300) Radeon 9700.
- Shader 3.0 - High Level Shader Language (2004/2005): Nvidia (NV40) GeForce 6, ATI (R520).
- Unified Shading Architecture (2006/2007): Nvidia (G80) GeForce 8800GTX, ATI (R600) – Radeon HD 2900XT.

History
Market Motivation

ATI R100 (2000)

Nvidia NV20 (2001)

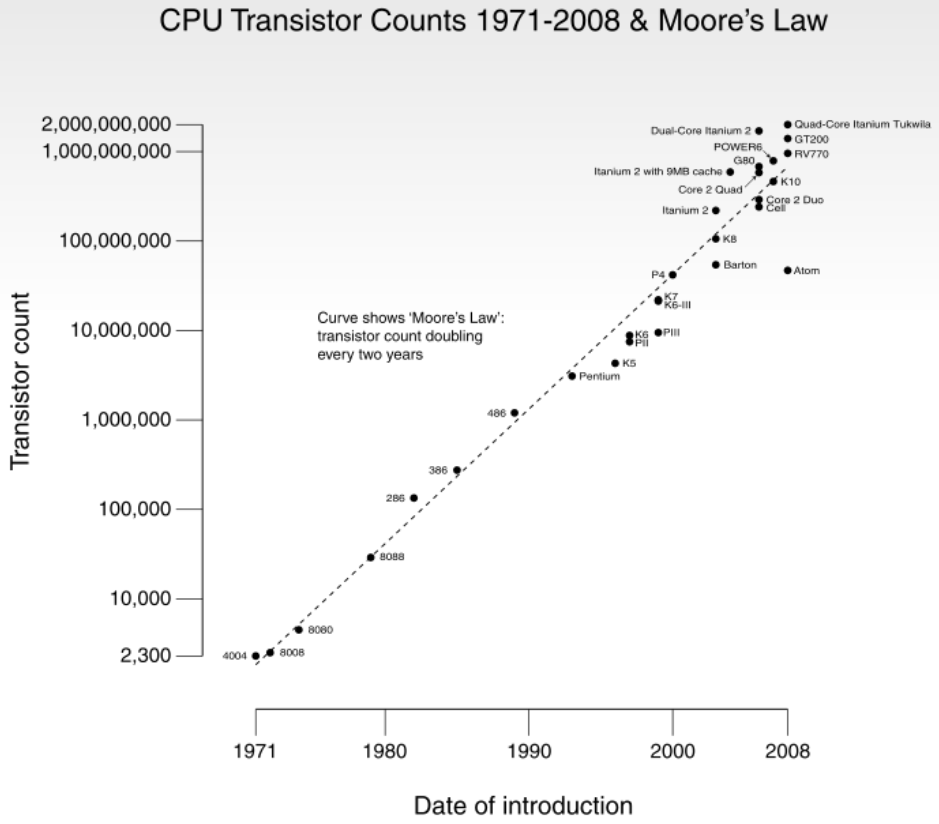
Nvidia NV40 (2004)

ATI R520 (2005)

Nvidia G80 (2006)

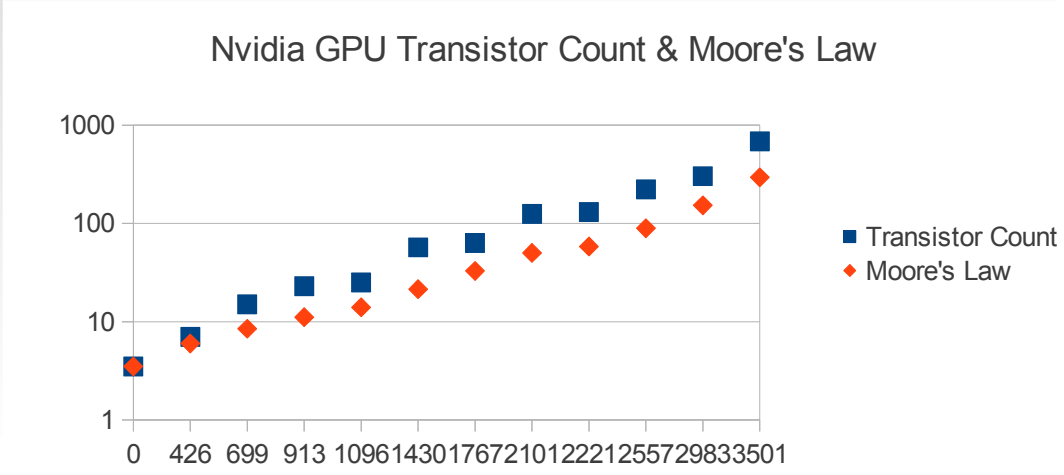
ATI R600 (2007)

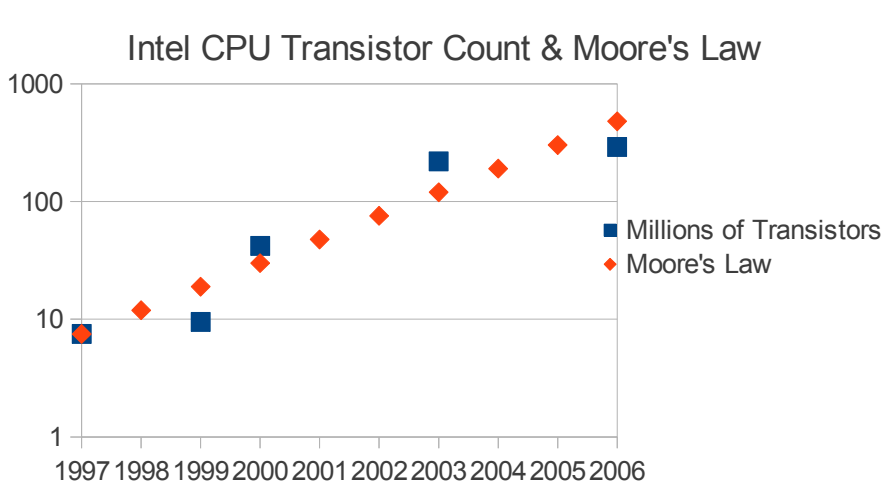
History
Technology Scaling

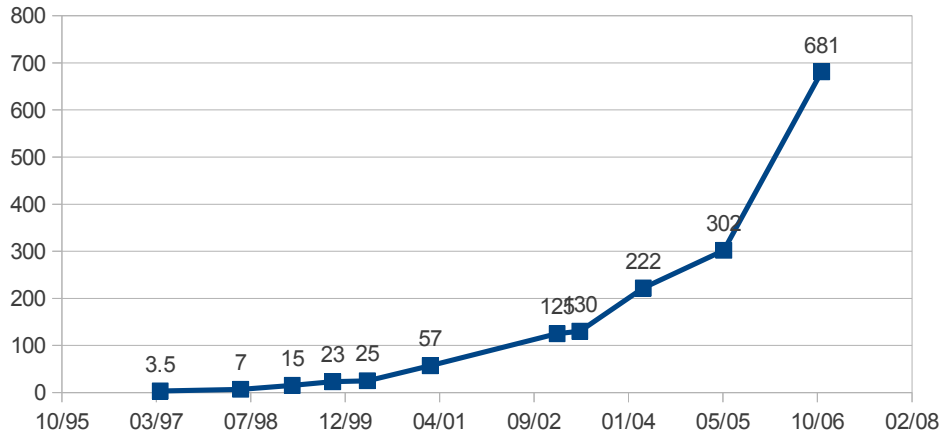


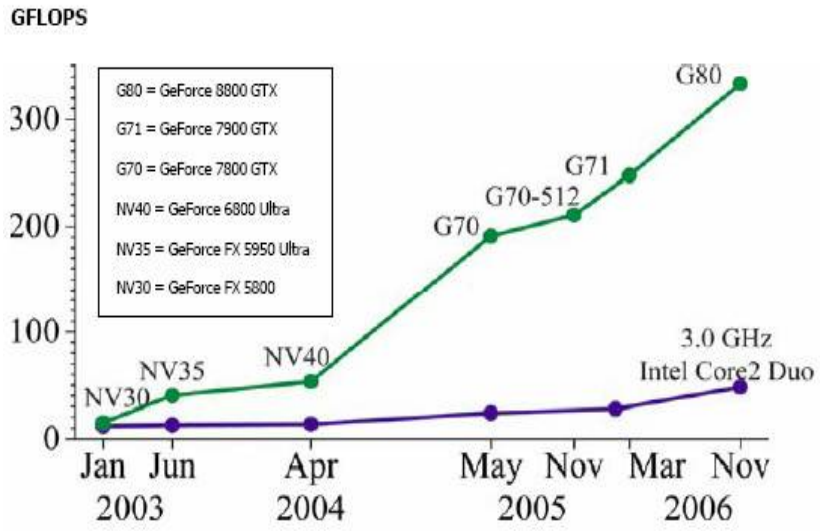
- Moore's Law: doubling of transistor count every 18 months inside a single device.

History
CPU Vs. GPU Scaling









Pioneering GPGPU Approach

After *Cg* language release (early 2002)

- Stream Processor $\rightarrow$ Fragment Processor
- Computing Kernel $\rightarrow$ Fragment Program (*Shader*)
- Output Stream $\rightarrow$ Group of rasterized Primitives
- Output Element $\rightarrow$ Rasterized Pixel
- Output stream is saved in *texture memory* and used as input for downstream kernels.
- OpenGL application, Shader written in *Cg* language.

Pioneering GPGPU
Promising results and useful hints

Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schr\&\#246;oder. 2003. **Sparse matrix solvers on the GPU: conjugate gradients and multigrid**. ACM Trans. Graph. 22, 3 (July 2003), 917-924. DOI=10.1145/882262.882364 http://doi.acm.org/10.1145/882262.882364

- Sparse Linear Algebra.
- Nvidia (NV30) GeForce FX, *Cg* shaders.
- 500 MHz GPU ~2 times faster than 3GHz Pentium 4
- Hints on how to improve *Gc* for scientific computing.

Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, John Spiletic, **Benchmarking and implementation of probability-based simulations on programmable graphics cards**, Computers & Graphics, Volume 29, Issue 1, February 2005, Pages 71-80, ISSN 0097-8493, 10.1016/j.cag.2004.11.

- Monte Carlo simulations
- Ising and Percolation models
- Measured 3.5 Gflops out of 16 Gflops peak

	Lattice size (not necessary power of 2)				
	128 × 128	256 × 256	512 × 512	1024 × 1024	2048 × 2048
GPU sec/frame	0.0007	0.0027	0.011	0.043	0.19
CPU sec/frame	0.0020	0.0069	0.028	0.116	0.55

GPU (NV30) and CPU (2.8 GHz Pentium 4) performances in processing a single step of the 2D Ising model.

GPGPU Devices: Nvidia Tesla

- 2007: starting from G80 family, Nvidia release the Tesla Unified Graphics and Computing Architecture

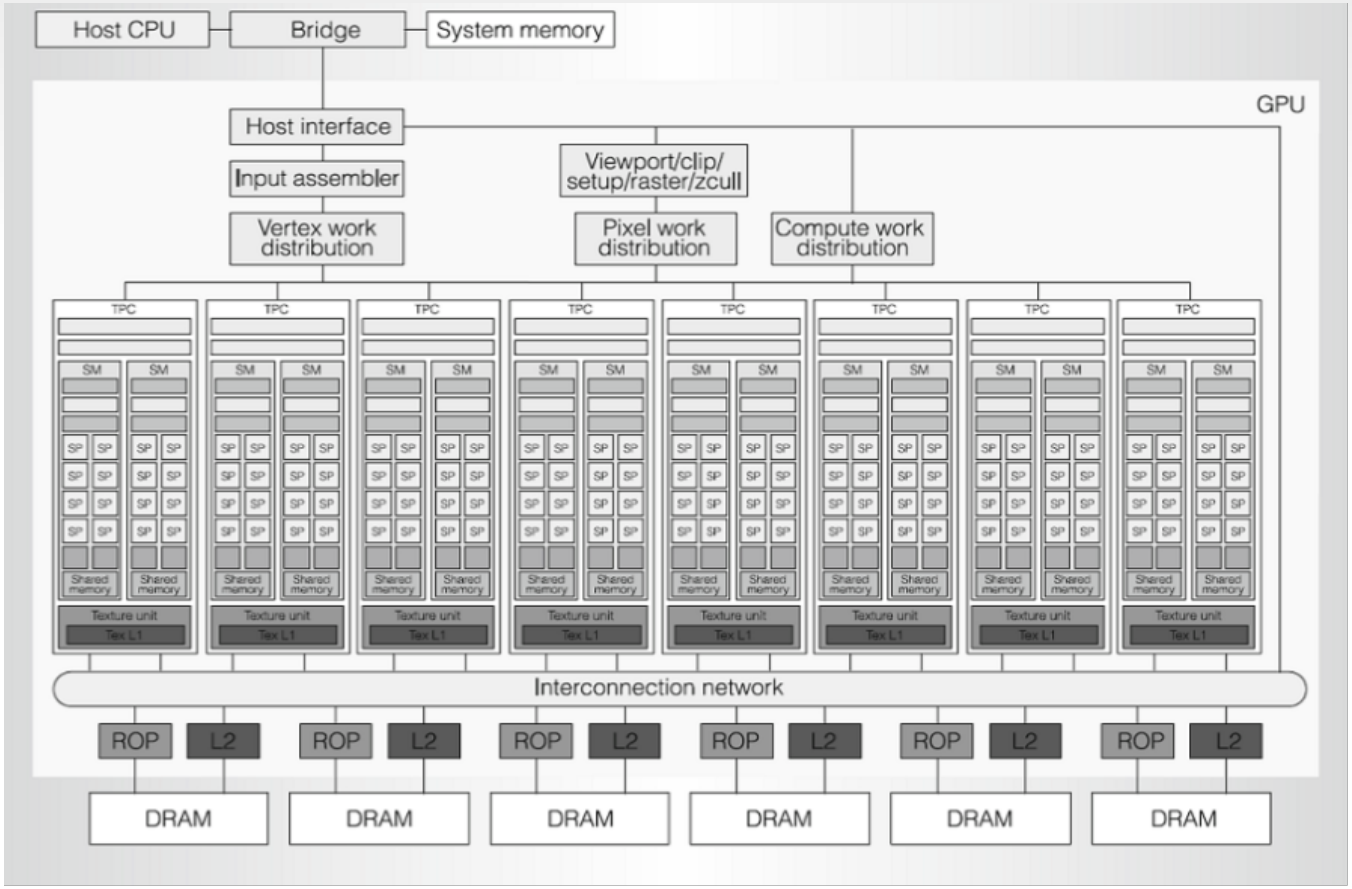
TPC: texture/processor cluster.

SM: streaming multiprocessor.

SP: streaming processor.

ROP: raster operation processor.

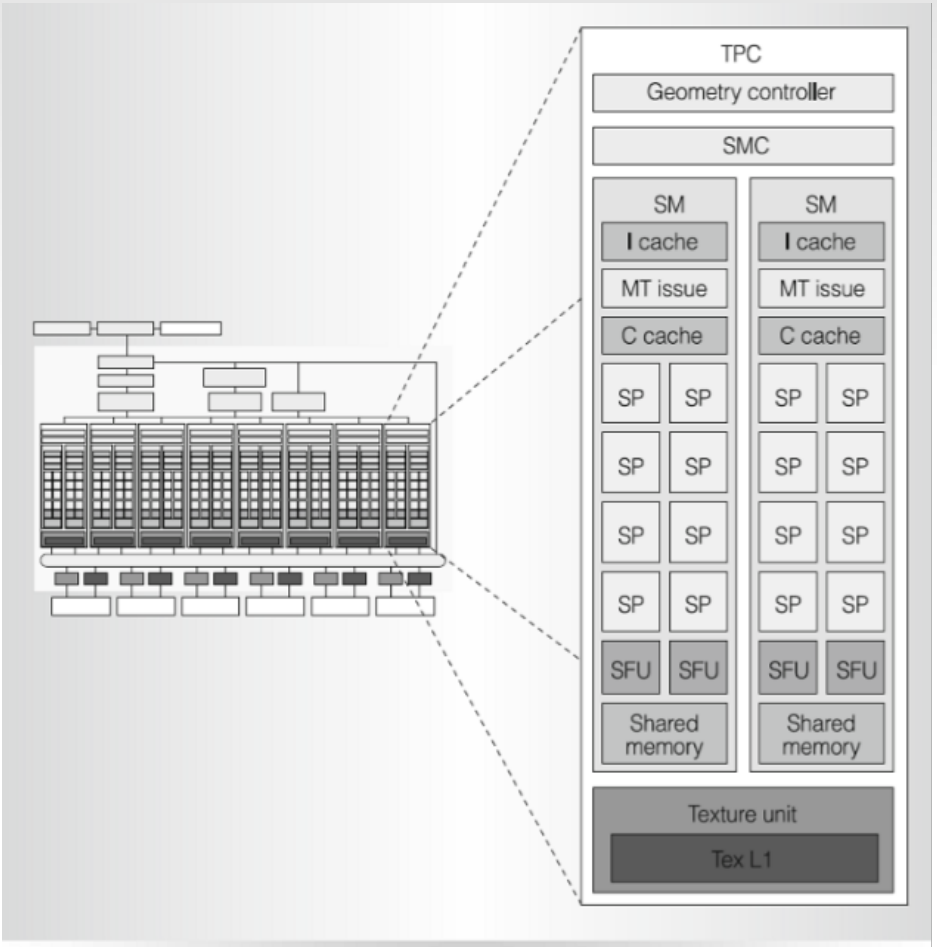
Tex: texture



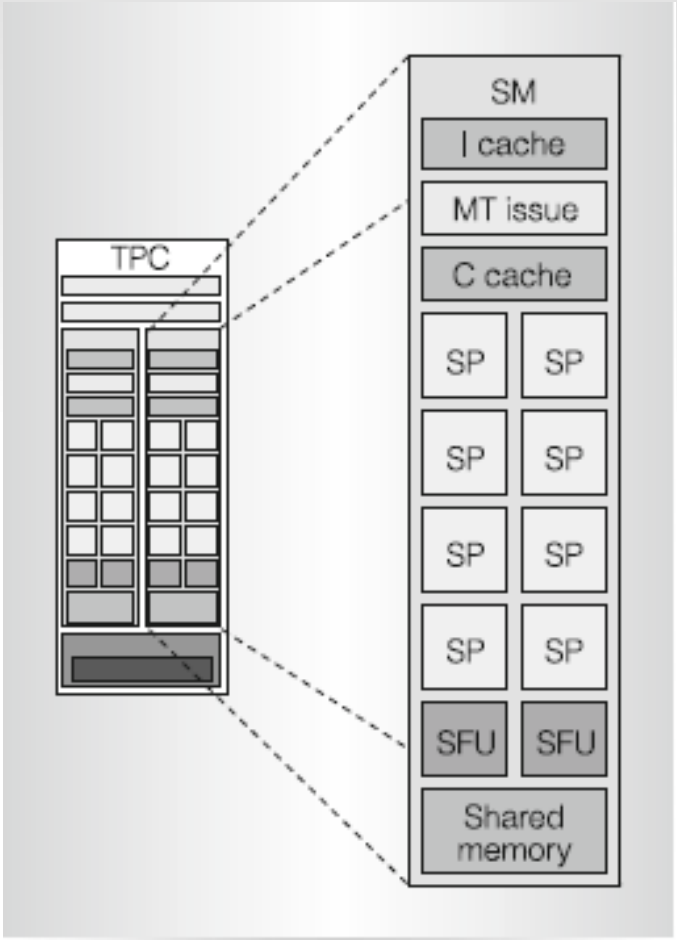
Lindholm, E.;Nickolls, J.;Oberman, S.; **NVIDIA Tesla: A Unified Graphics and Computing Architecture**; IEEE Micro Magazine - March-April 2008.

Nvidia Tesla Architecure (G80)

- Scalable processor array
  - 128 streaming processor (SP) cores distributed in
  - 16 streaming multiprocessor (SM) organized in
  - 8 indipendent texture/processor clusters (TPCs)



Streaming Multiprocessor Architecture (G80)



- Instruction cache.
- MT issue: multithreaded instr fetch & execute unit.
- Read only constant cache.
- 16 KB R/W shared memory.
- SP throughput: 1 multiply-add instruction per cycle.
- SFU: transcendental functions + 4 FP multipliers.

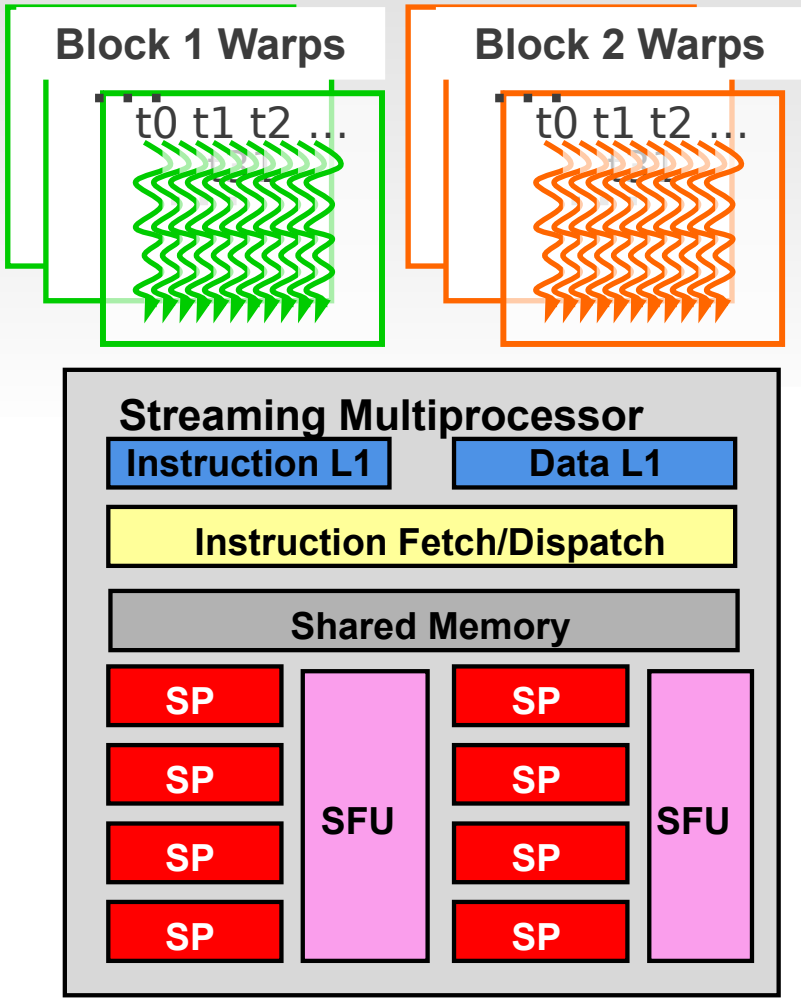
  @1.5 GHz $\rightarrow$ 36 Gflops.

Streaming Processor Multithreading (G80)

- HW support to manage up to **768** concurrent threads:
  - Lightweight thread creation.
  - Zero-overhead thread scheduling.
  - Fast barrier synchronization.
  - Each thread retains its own state and can follow an independent code path.
- SIMT model - Single Instruction Multiple Thread:
  - Threads are created and managed in groups of 32 (a ***warp***).
  - G80 SM manages up to 24 warps.
  - Warp threads start together from the same program address, but then may explore different code paths.
  - Maximum efficiency with minimum thread path divergence.

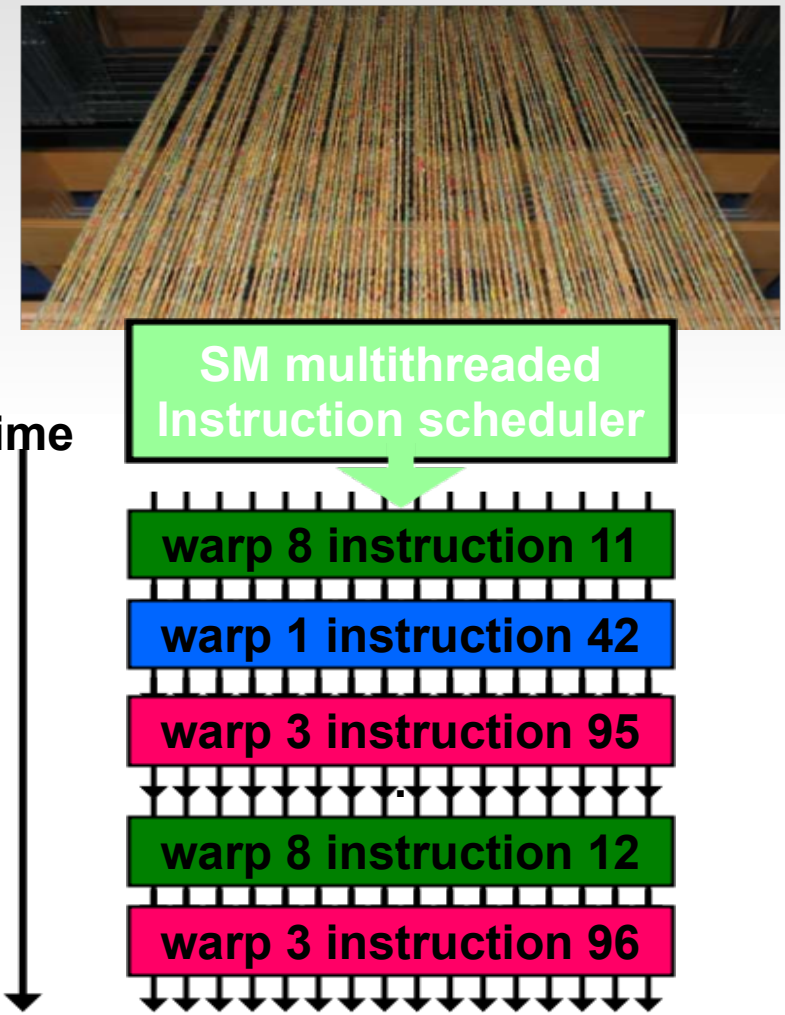
Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



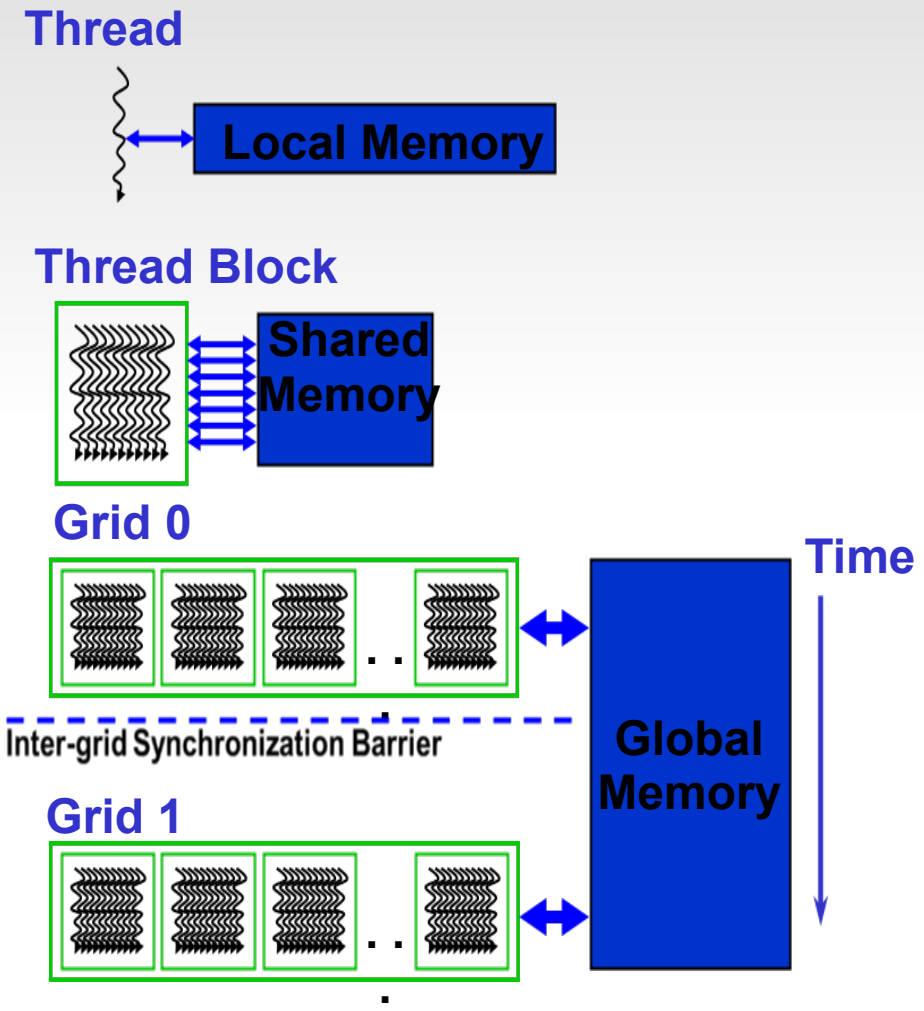
SM Warp Scheduling

- SM hardware implements zero overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution.
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency



Execution and Memory Granularity

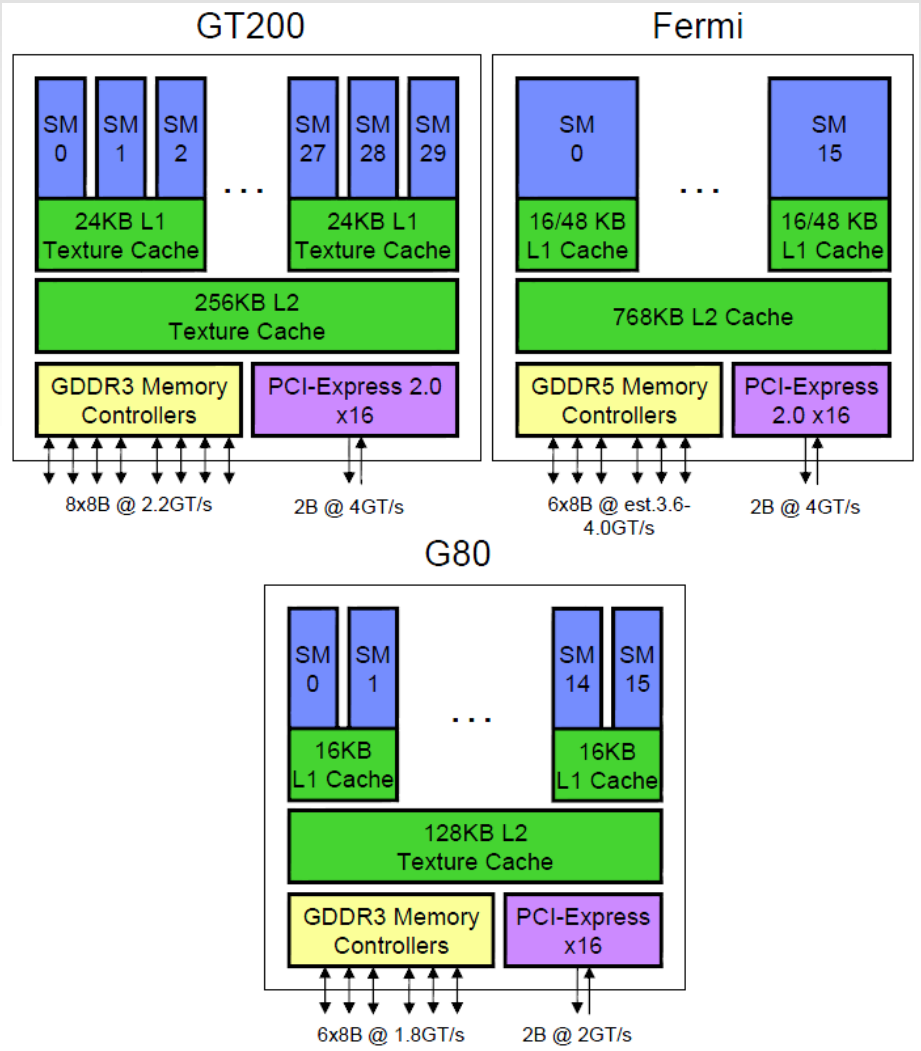
- Local memory: private per-thread memory for registers spill and stack.
- Shared memory: data sharing between threads in the same block
- Global memory: where sequential grids communicate and share large data sets.



Tesla GT200 & Fermi GF100 Devices

Configuration	Model	Archi-tecture	# of GPUs	Core clock (MHz)	Shaders		Memory					Processing Power (peak) GFLOPs[34]			Compute capability	TDP (watts)	Notes/Form factor
					Thread Processors (total)	Clock (MHz)	Bus type	Bus width (bit)	Memory (MB)	Clock (MHz)	Bandwidth (total) (GB/s)	Single Precision(SP) Total(MUL+ADD+SF)	Single Precision(SP) MAD(MUL+ADD)	Double Precision(DP) FMA			
GPU Computing Processor	C870[1]	G80	1	600	128	1350	GDDR3	384	1536	1600	76.8	518.4	345.6	0	1.0	170.9	Internal GPU (Full-height card)
GPU Computing Processor	C1060[2]	GT200	1	602	240	1300	GDDR3	512	4096	1600	102.4	933.12	622.08	77.76	1.3	187.8	Internal GPU (Full-height card)
M2050 GPU Computing Module	M2050	GF100	1	575	448	1150	GDDR5	384	3072[5]	3092	148.4	1288	1030.4[6]	515.2	2.0	225	Computing Module IEEE 754-2008 FMA capabilities
M2070/M2070Q[35] GPU Computing Module	M2070/M2070Q	GF100	1	575	448	1150	GDDR5	384	6144[5]	3132	150.336	1288	1030.4[6]	515.2	2.0	225	Computing Module IEEE 754-2008 FMA capabilities
M2090 GPU Computing Module	M2090	GF110	1	650	512	1300	GDDR5	384	6144[5]	3700	177.4	1664	1331.2[6]	665.6	2.0		Computing Module IEEE 754-2008 FMA capabilities

Tesla GT200



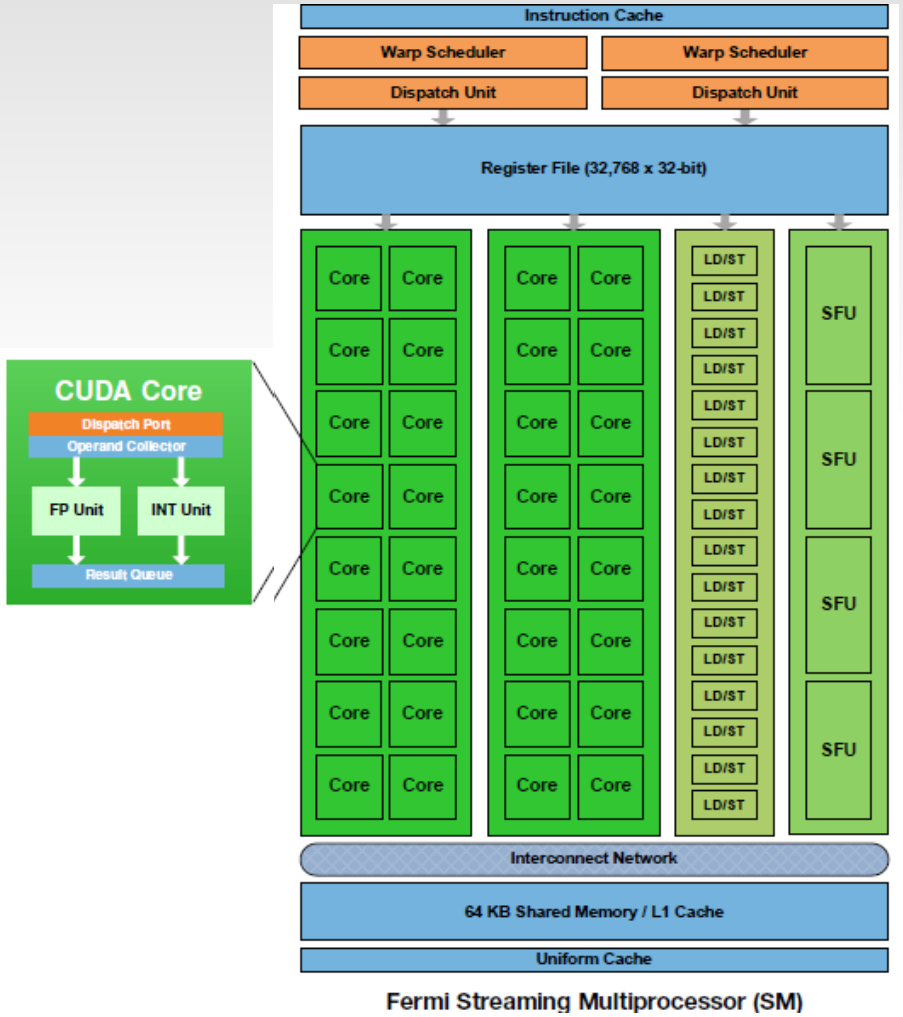
- GT200:
  - 30 SM, each:
    - 8 SP, 1 DP
    - 64KB Register File
    - 16KB Shared Mem
    - 32 Entry Warp Scheduler
    - A thread can reserve 4-128 32 bits registers
    - 8 FMADs and 8 FMULs per cycle

Fermi GF100

- 3 billion transistors
- 512 CUDA Cores
- Better DP performance
- ECC Memory support
- L1 & L2 Cache
- Concurrent Kernels (up to 16)
- Faster Context Switching (10x)
- Unified Address Space, enabling C++

Fermi GF100
SM Architecture

- 32 CUDA cores
- 128 KB Register File
- 48 or 16 KB of Shared Memory
- 16 or 48 KB of L1 cache
- FMA (single and double precision) IEEE 754-2008
- Re-designed integer ALU optimized for 64 bit operations
- Dual Warp Scheduler



Fermi Streaming Multiprocessor (SM)

Development Tools

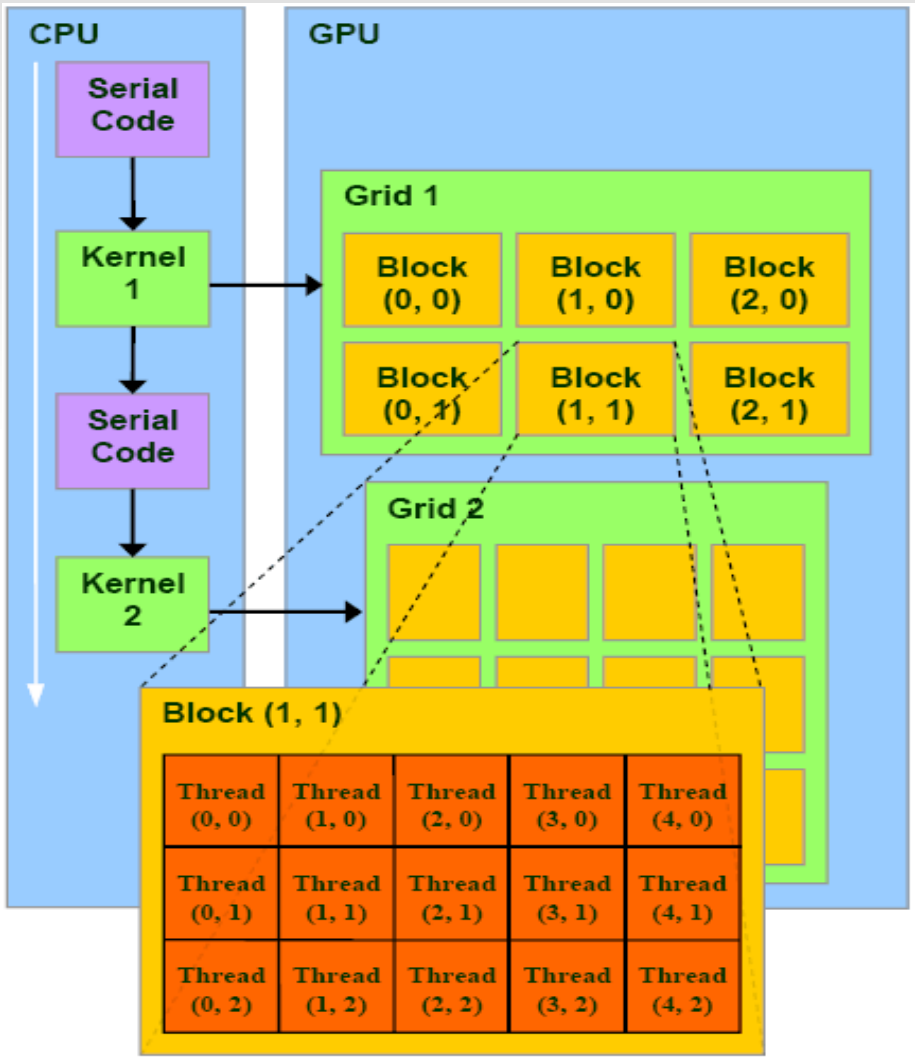
- 2004: brook streaming language (Nvidia, ATI)
- 2007: Nvidia CUDA
- 2008: OpenCL()

  You don't necessarily need to learn a new language to start using GPUs:

- Numerical Packages: MATLAB, Mathematica,...
- Libraries
- C CUDA and libraries examples

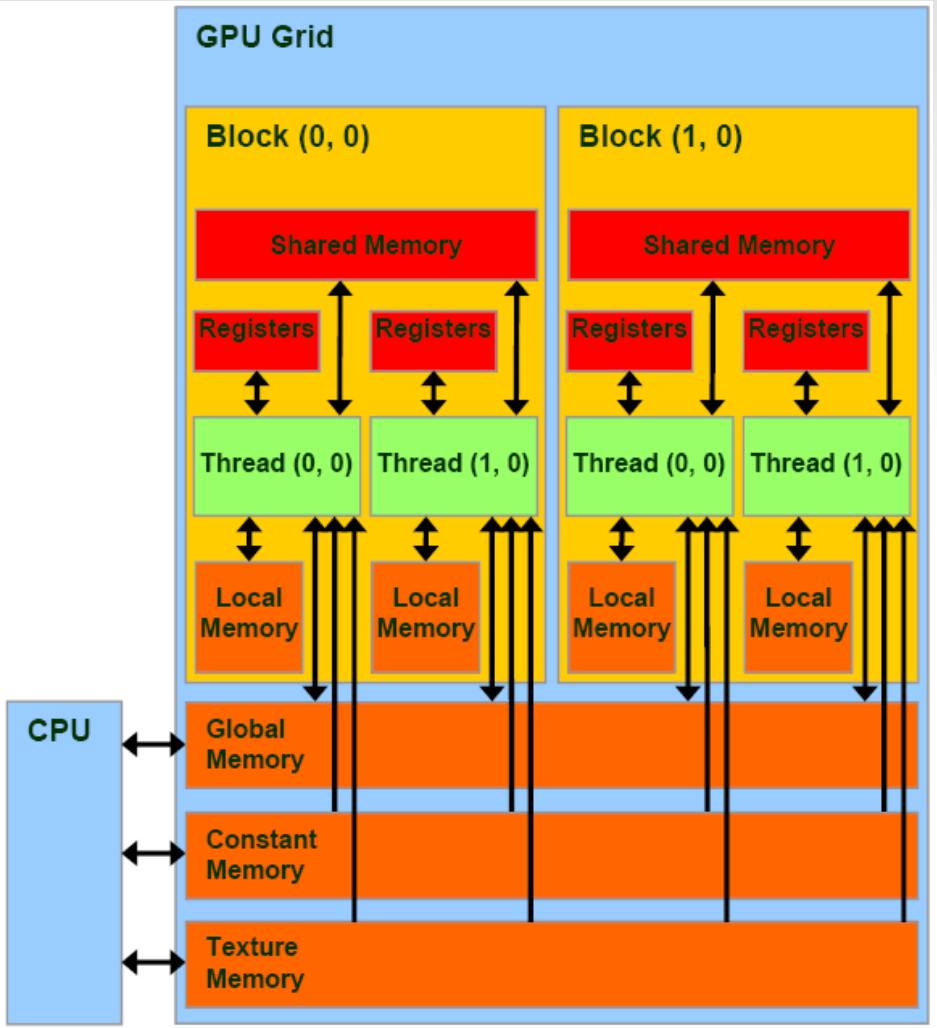
CUDA Programming Model
Expressing Parallelism

- built around a scalable array of multithreaded Streaming Multiprocessors (SMs)
- 2D Grid of thread blocks (coarse grain parallelism)
- 3D array of threads in a block (fine grain parallelism)
- Thread blocks are distributed to Stream Multiprocessors for execution (possibly many)
- Threads belonging to the same block get executed concurrently in the SM.



CUDA Memory Model

- cpu/gpu code different access to memories
- CUDA API
  - cudaMalloc() on cpu: allocates objects in Gpu global memory
  - cudaFree()
  - cudaMemcpy()
    - CPU to GPU
    - GPU to CPU
    - CPU to CPU
    - GPU to GPU



CUDA C Example

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N) C[i] = A[i] + B[i];
}

// Host code
int main()
{ int N = ...;
  size_t size = N * sizeof(float);
  // Allocate input vectors h_A and h_B in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  // Initialize input vectors ...
  // Allocate vectors in device memory
  float* d_A;
  cudaMalloc(&d_A, size);
  float* d_B;
  cudaMalloc(&d_B, size);
  float* d_C;
  cudaMalloc(&d_C, size);
```

```
  // Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // Free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  // Free host memory ...
}
```

`__global__`: kernel definition
`blockIdx, threadIdx`: built_in variables
`<<<dimgrid,dimblock>>>`: 2D dimension of grid of blocks, 3D dimension of thread array

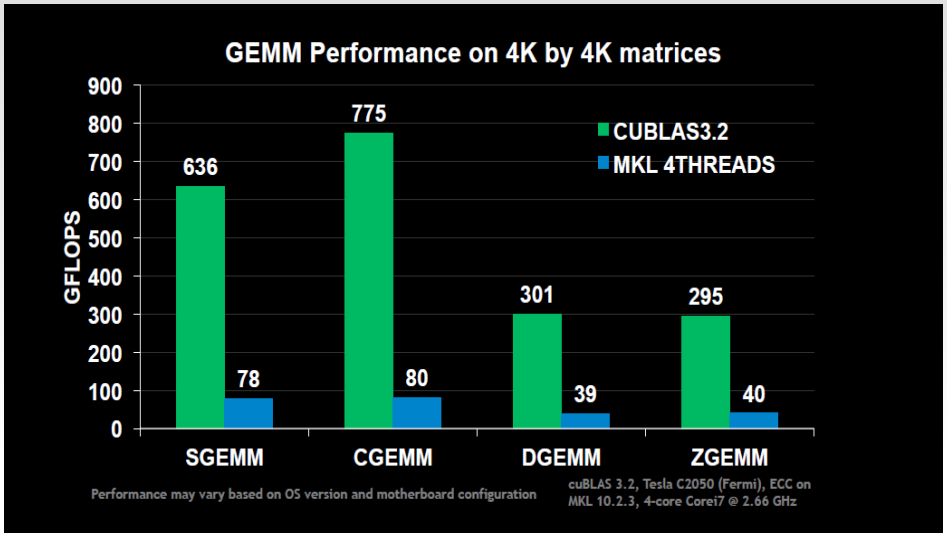
Libraries

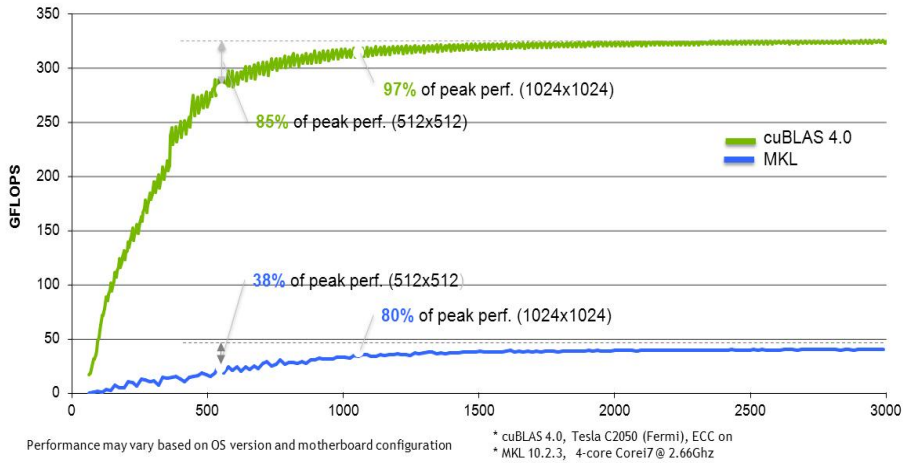
CUDA Toolkit includes several libraries:

- Standard C Math Library (**LIBM**)
- Dense Linear Algebra (**cuBLAS**)
- Sparse Linear Algebra (**cuSPARSE**)
- Pseudo and Quasi random number generators (**cuRAND**)
- Fast Fourier Transform (**cuFFT**)
- Image & Signal Processing (**NPP**)
- STL-like Parallel Algorithms Template Library (**Thrust**)

Linkable from C/C++ and Fortran.

cuBLAS





- Implementation of BLAS (Basic Linear Algebra Subprograms).
- Single, double, complex and double complex data types (S, D, C, Z).
- All 152 standard BLAS routines.
- Column-major storage.
- Helper functions (memory allocation, data transfer).
- Support for CUDA streams.
- V4.0 supports multiple GPUS and concurrent kernels.

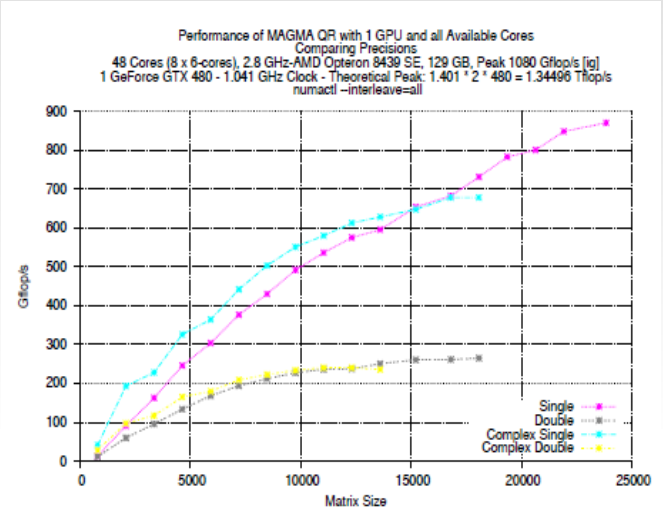
Libraries

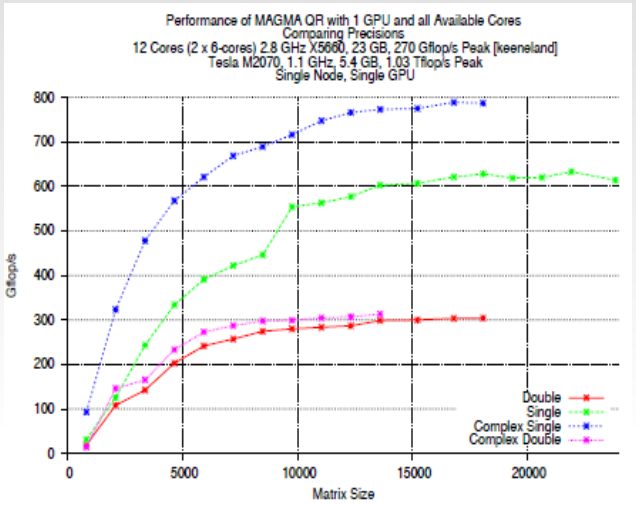
Several open source and commercial* libraries:

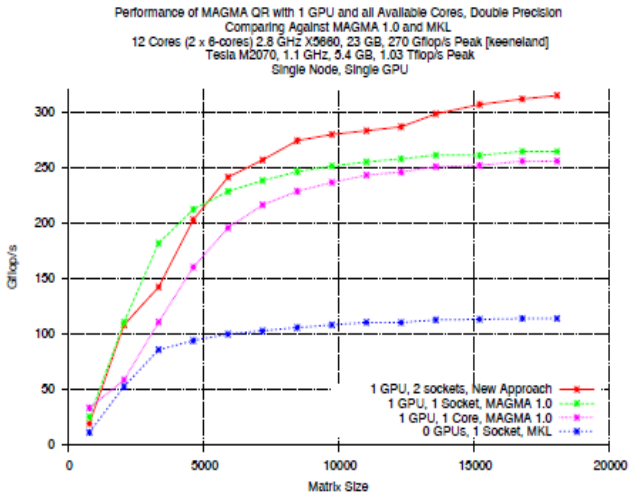
- MAGMA: Linear Algebra
- CULA Tools*: Linear Algebra
- OpenVidia: Computer Vision
- OpenCurrent: CFD
- CUSP: Sparse Linear Solvers
- Gromacs, AMBER, NAMD: molecular dynamics
- CUDA-meme: gene sequencing
- NAG*: Computational Finance
- Many others...
- Surprisingly: GPUs are very good for search algorithms: availability of hw resources lead to new algorithmic developments exploiting them

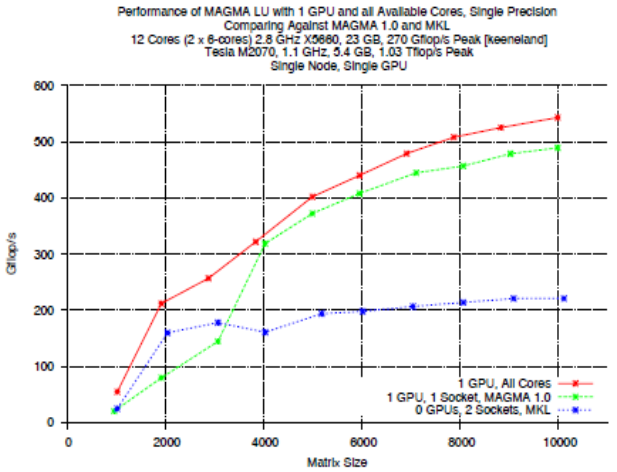
MAGMA: Matrix Algebra for GPU and Multicore Architectures

- LAPACK style
- multicore+GPU systems
- heuristic autotuning: generation of multiple code variants, selecting the fastest ones through benchmarking.
- QR, LU factorization



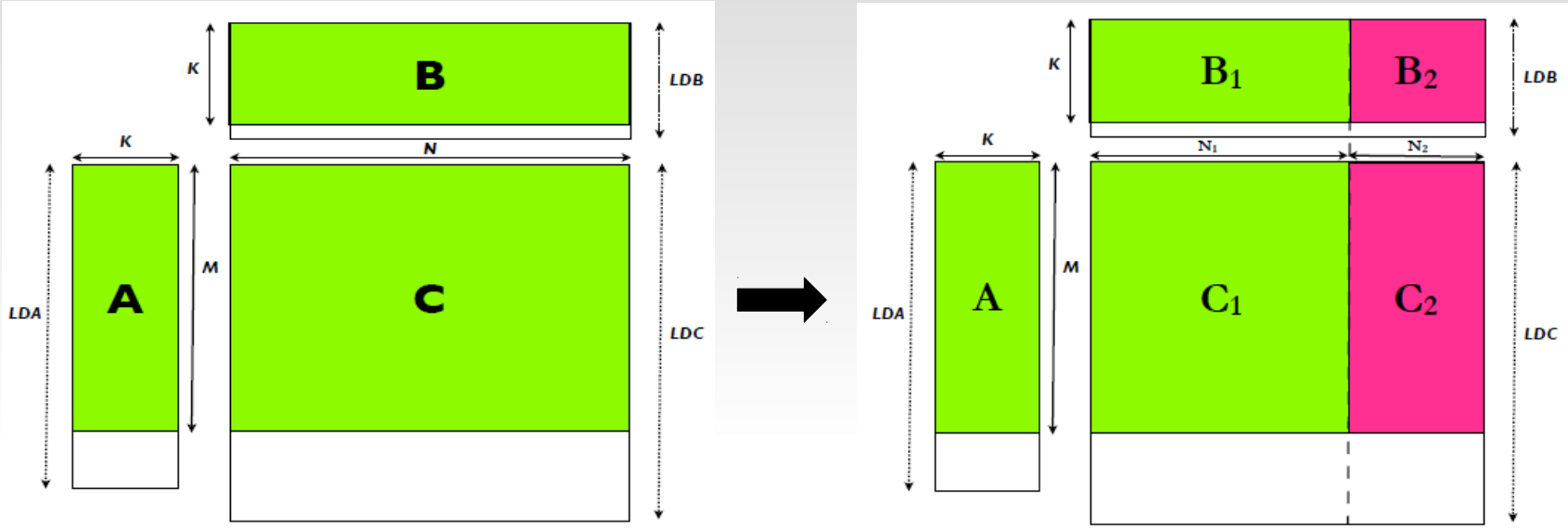






Horton, M., Tomov, S., Dongarra, J. "A Class of Hybrid LAPACK Algorithms for Multicore and GPU Architectures," Symposium for Application Accelerators in High Performance Computing (SAAHPC'11), Knoxville, TN, July 19-20, 2011.

CPU/GPU Task Partitioning



- $C \leftarrow \alpha A B + \beta C$
- $DGEMM(A, B, C) = DGEM(A, B_1, C_1) \cup DGEMM(A, B_2, C_2)$
- $T_{GPU}(M, K, N_1) = 8(KM + KN_1 + MN_1)/B_{H2D} + 2KMN_1/G_{GPU} + 8MN_1/B_{D2H}$
- $T_{CPU}(M, K, N_2) = 2KMN_2 / G_{CPU}$

GPU

CPU

$O(N^2)$ $O(N^3)$

Massimiliano Fatica. 2009. Accelerating linpack with CUDA on heterogenous clusters. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2). ACM, New York, NY, USA, 46-51. DOI=10.1145/1513895.1513901 http://doi.acm.org/10.1145/1513895.1513901

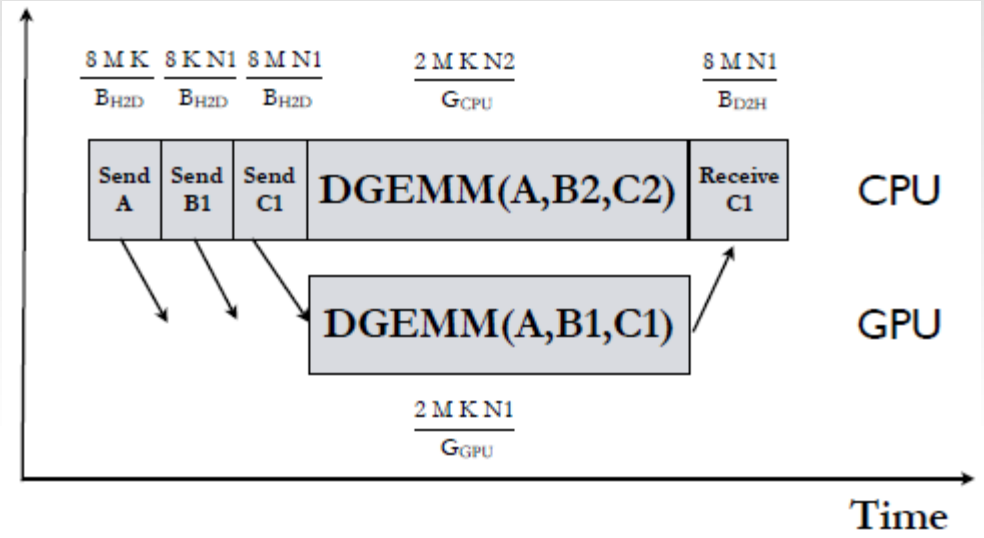
CPU/GPU Task Partitioning

- Optimal Partitioning:

  $T_{GPU}(M, K, N_1) = T_{CPU}(M, K, N_2)$

- Omitting $O(N^2)$ data transfer:

  $N_1/N = G_{GPU}/(G_{CPU}+G_{GPU})$




```
// Copy A from CPU memory to GPU memory devA
status = cublasSetMatrix (m, k , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1 from CPU memory to GPU memory devB
status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1 from CPU memory to GPU memory devC
status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);
// Perform DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);
// Perform DGEMM(A,B2,C2) on CPU
dgemm_cpu('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);
// Copy devC from GPU memory to CPU memory C1
status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```

Lattice QCD

C. Bonati (Pisa), G. Cossu (KEK), M. D'Elia (Genova),
A. Di Giacomo (Pisa) P. Incardona (Genova)

The physical problem: QCD and confinement

Low energy QCD and confinement are intrinsically nonperturbative phenomena. In Lattice QCD a finite lattice is introduced as a nonperturbative gauge invariant regulator and observables are calculated by using Monte Carlo simulations.

The numerical problem

In a LQCD simulation a lot of $L \times L$ linear systems have to be solved, with $L \sim 10^5 \div 10^6$

$\Rightarrow$

Need for
dedicated machines
(apeNEXT, Blue Gene, ...)

Lattice QCD

GPUs turned out to be low cost alternative to dedicated machines

With our C++/CUDA code, we have (depending on the parameters)

$$\text{1cpu} + \text{1C2050} \sim \begin{array}{l} 2 \div 6 \text{ apeNext} \\ \text{crate} \end{array} \sim \begin{array}{l} 70 \div 140 \text{ times} \\ \text{faster than} \\ \text{1 Opteron core} \end{array}$$

$$\sim \begin{array}{l} 40 \div 90 \text{ times faster than} \\ \text{1 Xeon X5560 core} \end{array}$$

$$\text{1cpu} + \text{1C2050} \sim \text{€}\,2000$$

Our strategy: all computations on GPU.
More details on algorithm and performance reported in arXiv:1106.5673

Lattice QCD

We have:

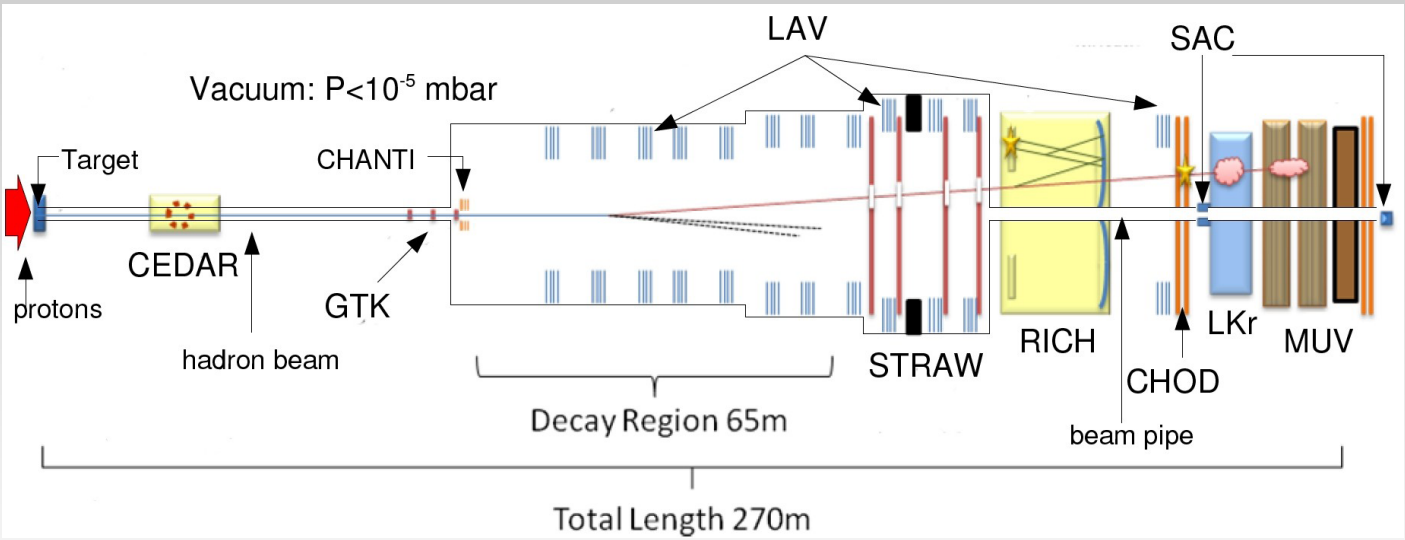
- This code is already used in production runs on the study of the QCD phase diagram (e.g. Phys. Rev. D **83**, 054505 (2011), arXiv:1011.4515).
- openCL support (by now less efficient than CUDA by $\sim 20\%$).

Current developments:

- Multi-GPU version: initial version working. Preliminary test: 4 GPU gain factor ~ 3.3.

Wish list:

- improved discretization of the action.



· The NA62 at CERN aims at measuring the Branching Ratio of the ultra-rare decay K→πνν(bar) (BR~10-10)

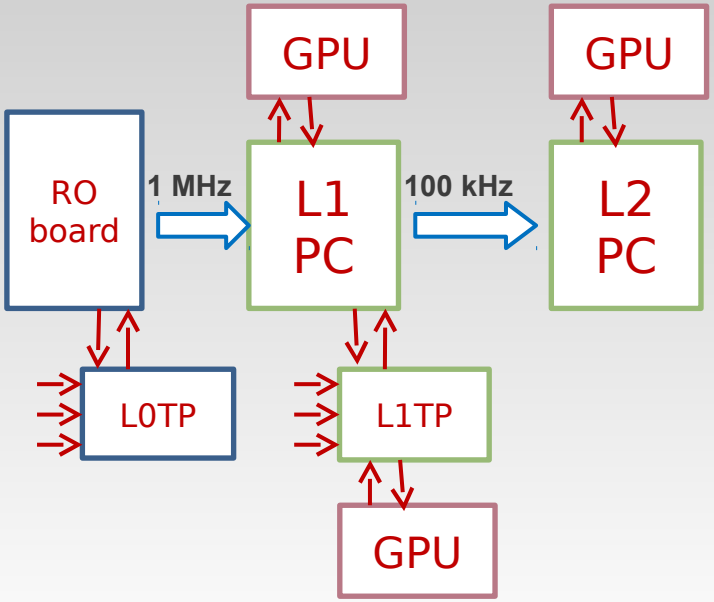
· The experiment requires a selective and efficient online selection for the events of interest for the measurement.

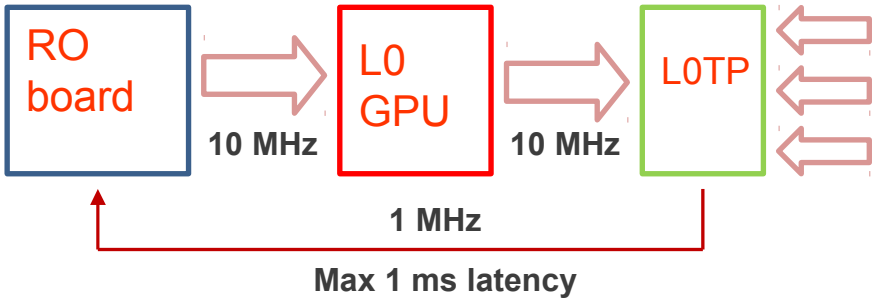
· The trigger is structured in 3 levels (L0-L1-L2): the first level is synchronous (hardware) with a fixed latency (1 ms), while the other levels are implemented in software.

· The initial rate of 10 MHz has to be reduced to 10 kHz for final acquisition on tapes.

[G.Lamanna, M.Sozzi]

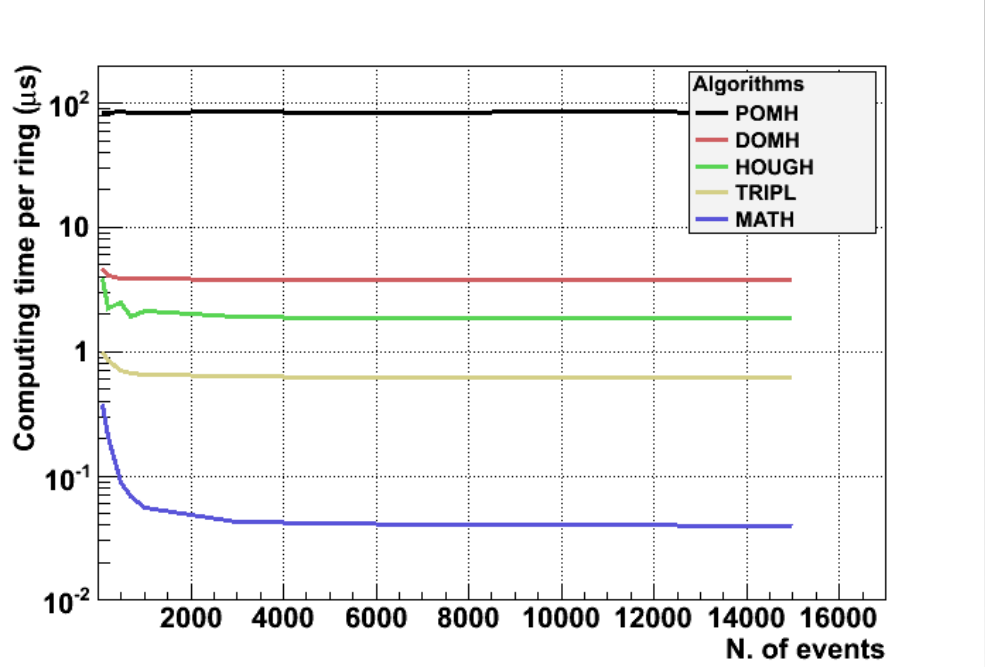
The use of GPUs to build “high quality” primitives can be exploited to define highly selective trigger conditions.





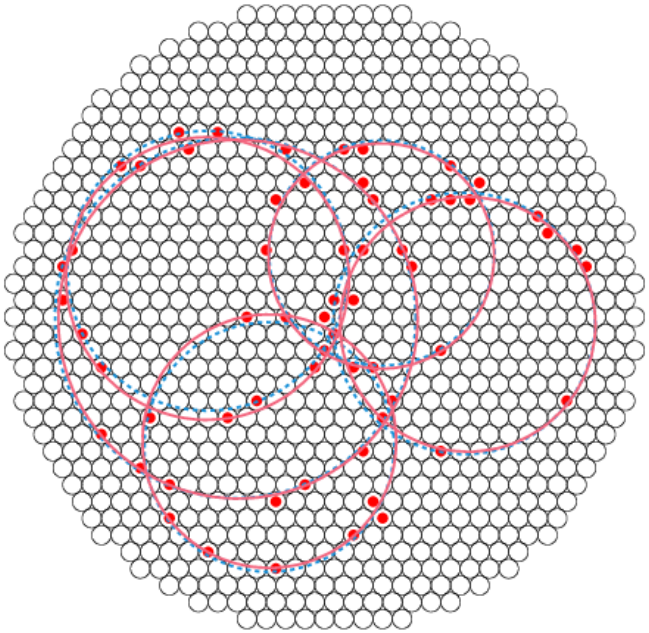
- In the software levels, based on PC, the use of GPUs is “trivial”.
- In the hardware level it’s important to have a small and stable latency, in order to ensure a “real-time” processing.

[G.Lamanna, M.Sozzi]



- As first exercise we implemented a fast rings pattern recognition for the RICH detector.

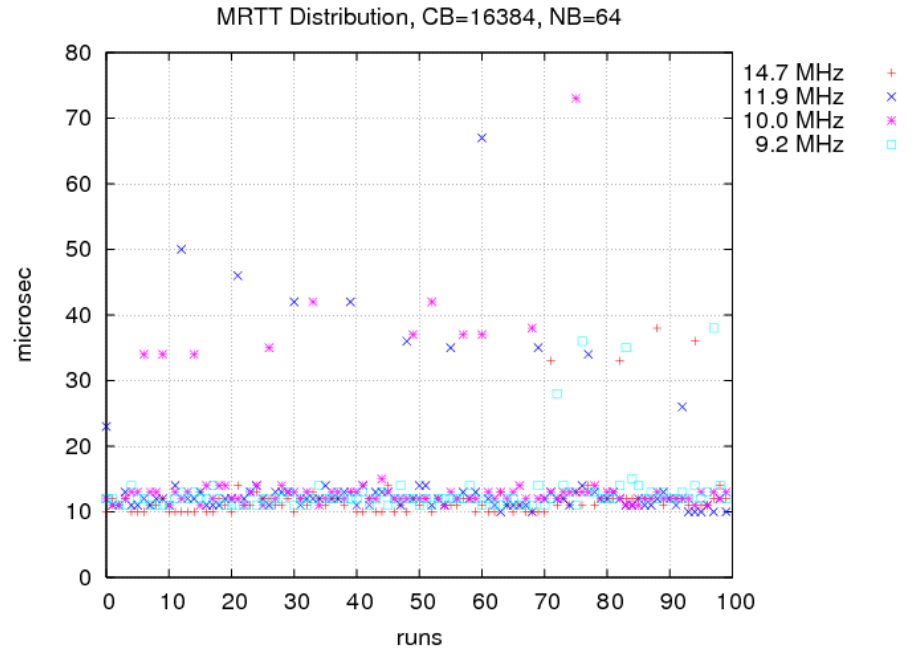
The faster algorithm for the L0, in TESLA C1060, identify a single tin in 50 ns
At L1 the multiple rings search need few tens of us, using a special parallel algorithm (called Almagest) developed for optimization on GPU.



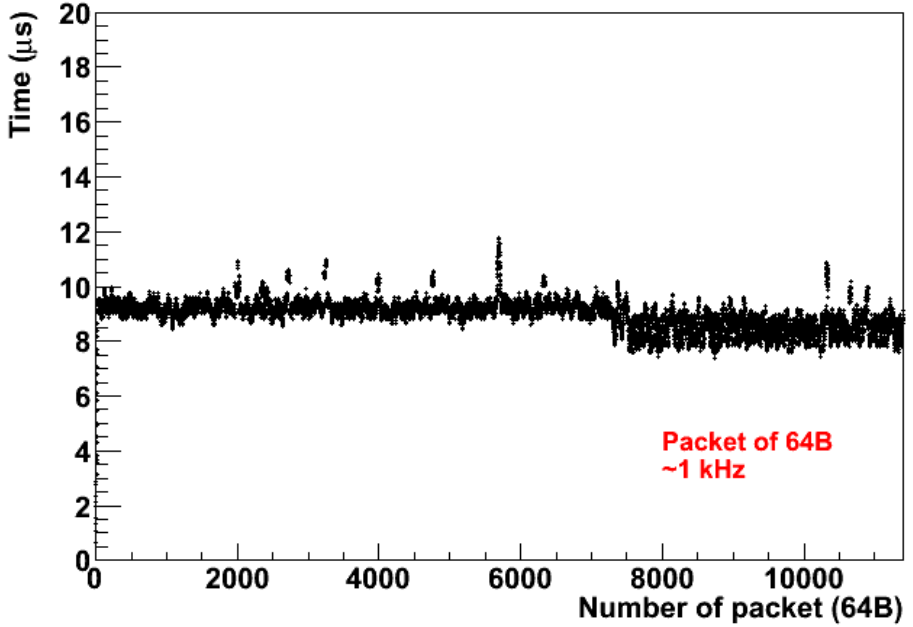
[G.Lamanna, M.Sozzi]

Since the L0 works in “real-time” all the contributions to the total latency have to be considered:

- Buffering time
- Transfer time through ethernet
- Transfer time from NIC to RAM
- Copy of the data between RAM and GPU
- Processing time
- Copy of the results from GPU to RAM

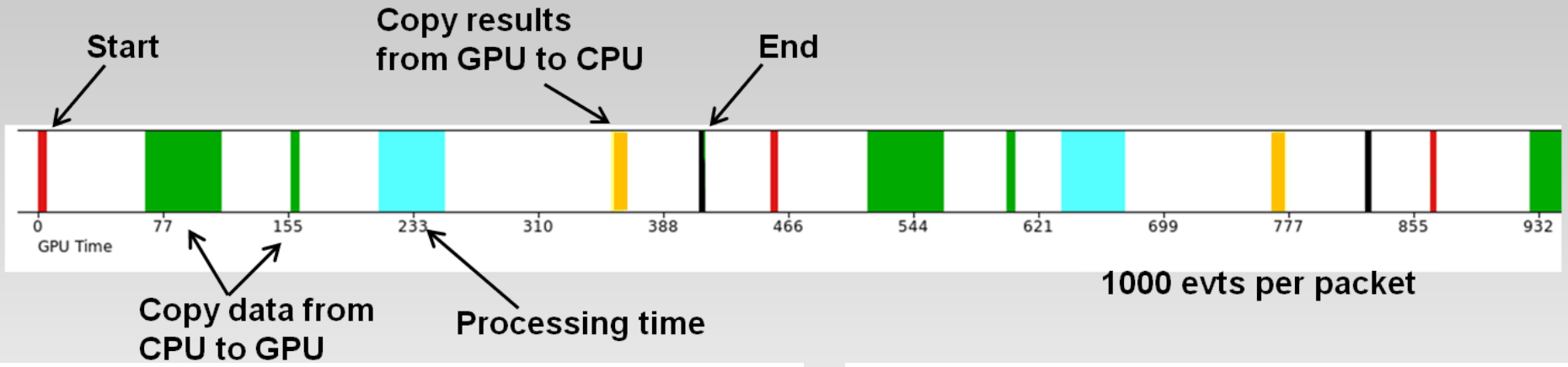


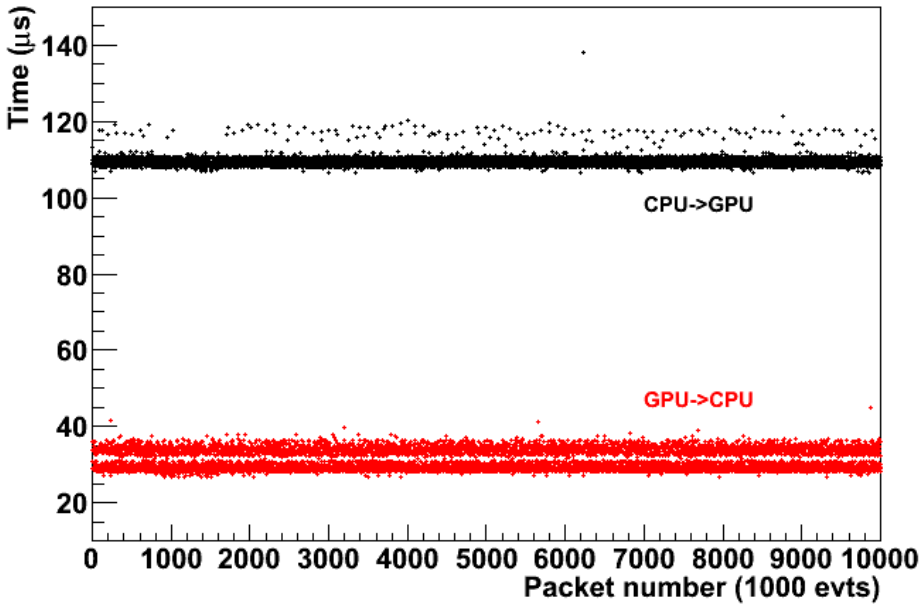
(Time from NIC to RAM)

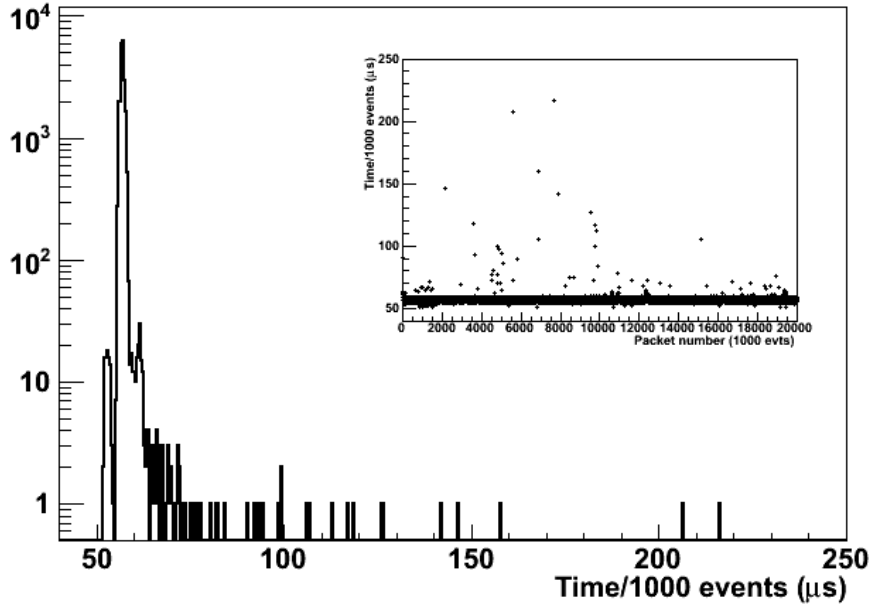


(Packet processing time in linux kernel)

[G.Lamanna, M.Sozzi]







The stability of transfer and execution times is studied in order to understand the deterministic behaviour of the system

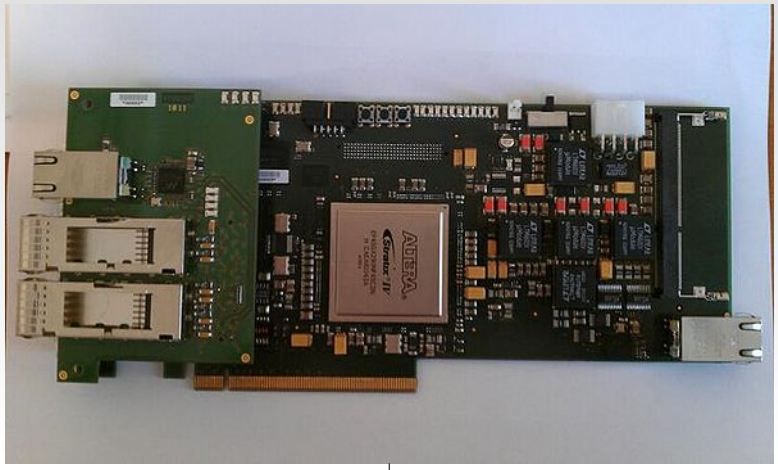
[G.Lamanna, M.Sozzi]

A cluster for LQCD...

APE group

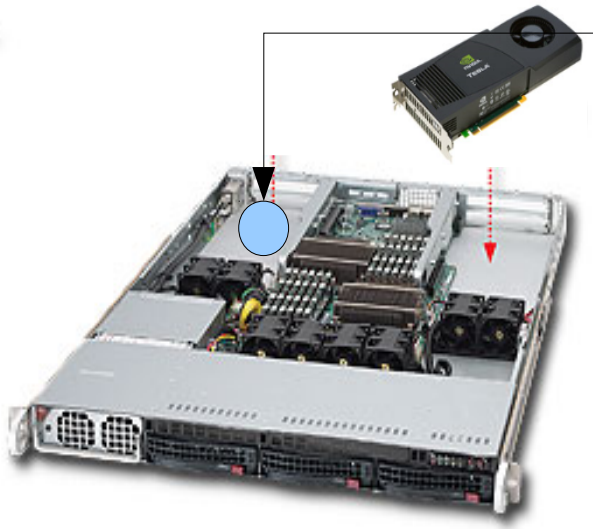
Our GPU cluster node:

- A dual-socket multi-core CPU
- 2 Nvidia M20XX GPUs
- one APEnet+ card

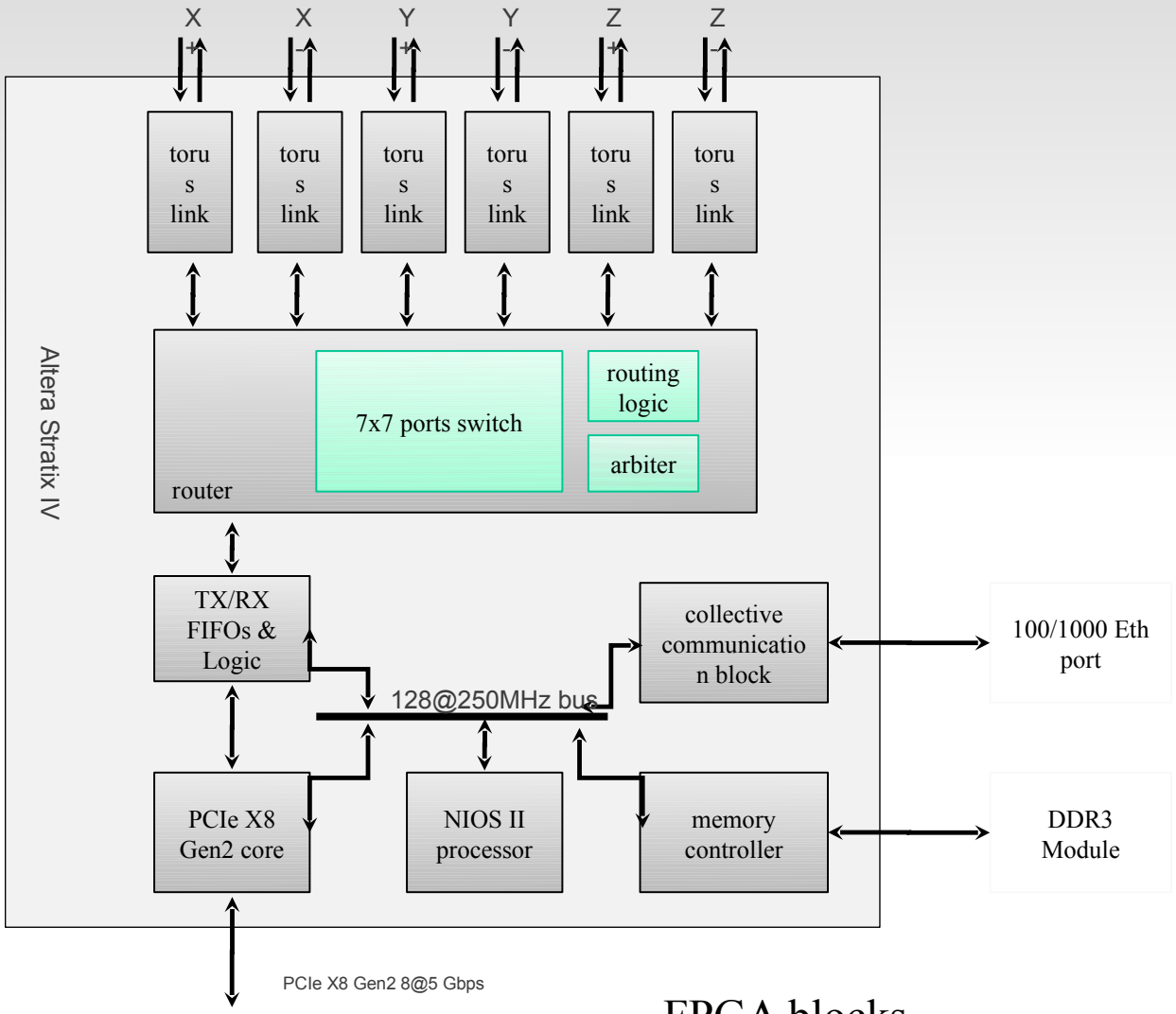


Our case study:

- 64^3x128 lattice
- Wilson fermions
- SP



APEnet+ HW



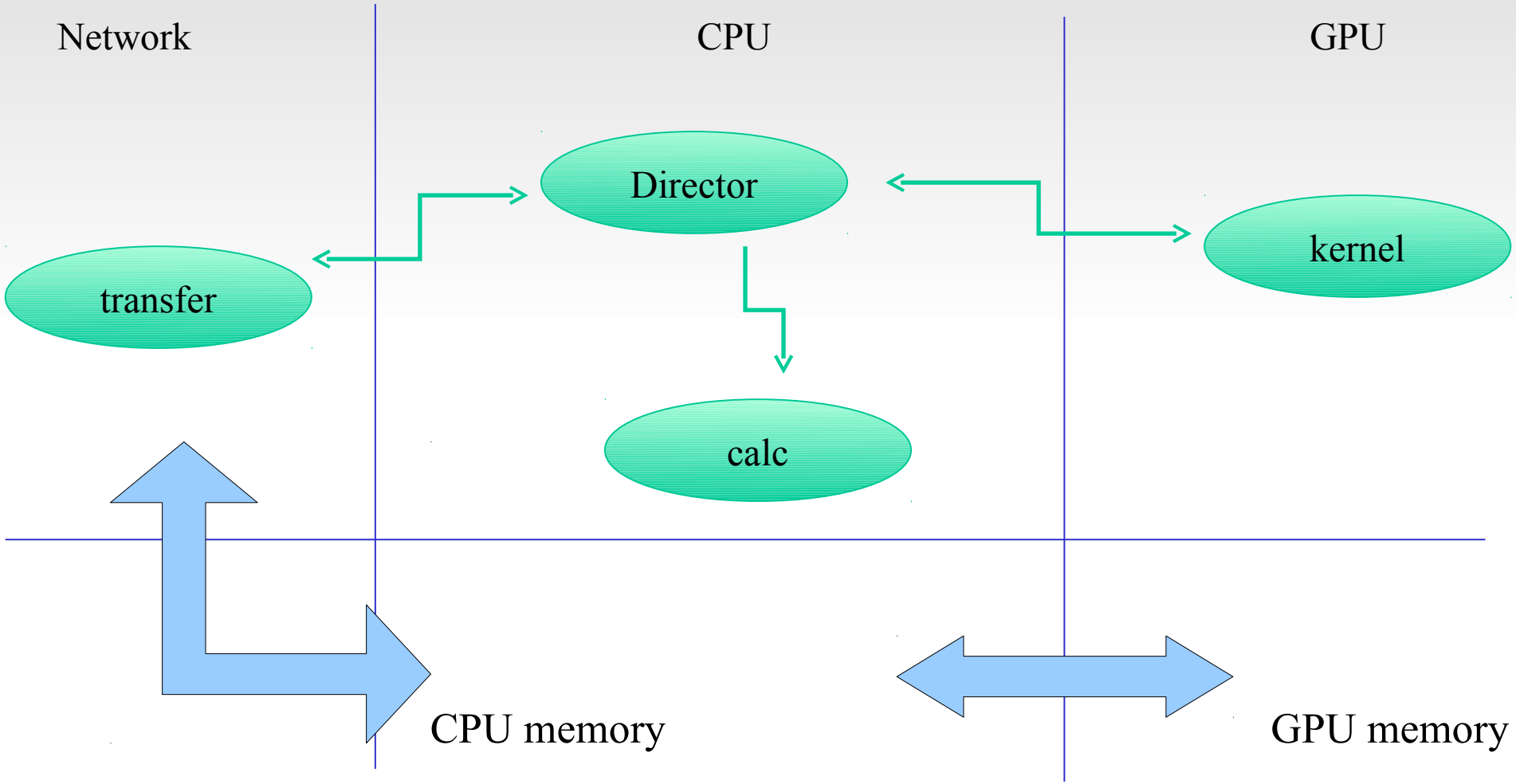
FPGA blocks

APE group

- 3D Torus, scaling up to thousands of nodes
  - packet auto-routing
  - 6 x 34+34 Gbps links
  - Fixed costs: 1 card + 3 cables
- PCIe X8 gen2
  - peak BW 4+4 GB/s
- A *Network Processor*
  - Powerful zero-copy RDMA host interface
  - On-board processing
  - Experimental direct GPU interface
- SW: MPI (high-level), RDMA API (low-level)

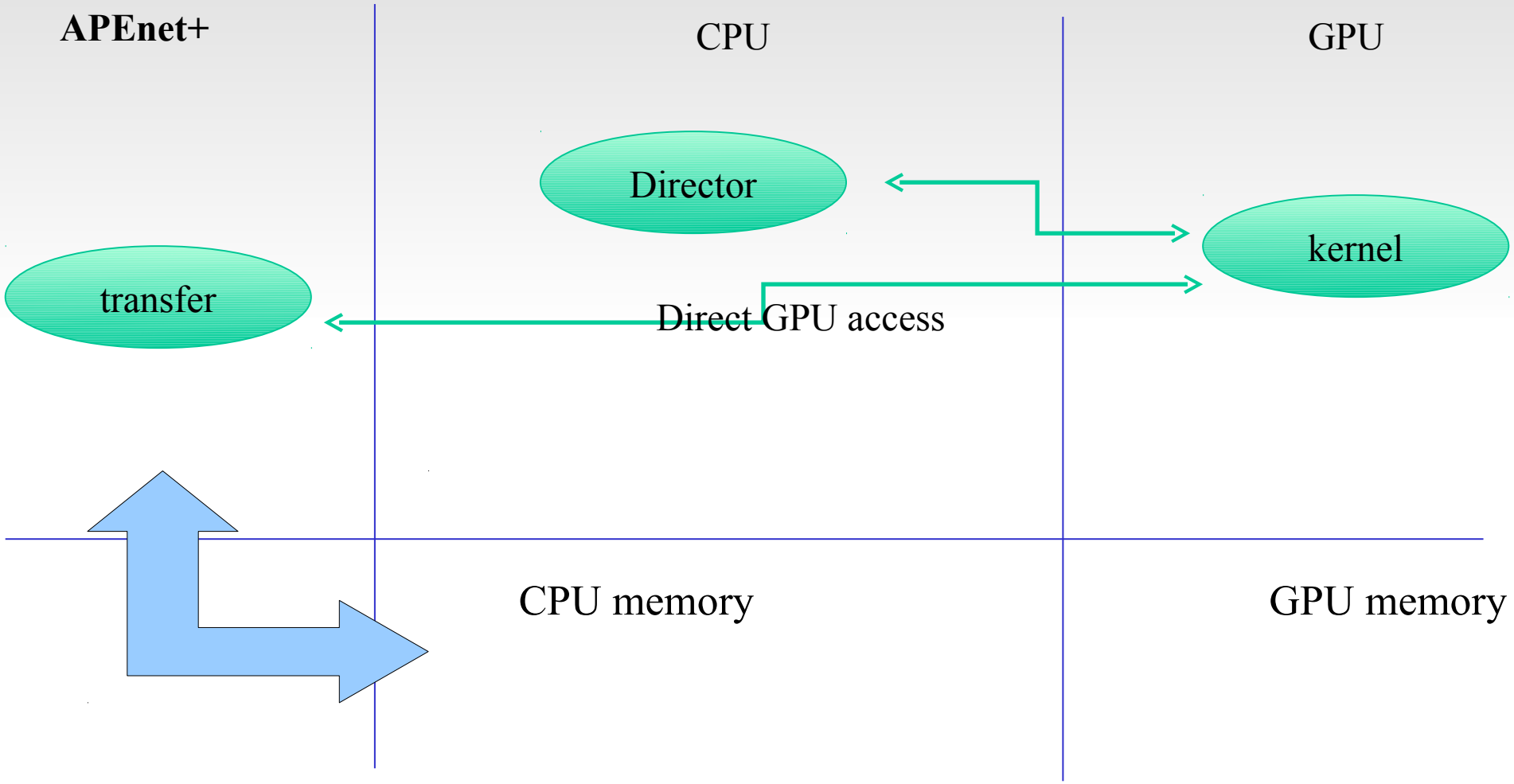
The traditional flow

APE group



Optimized network

APE group



Summary

- Heterogeneous ManyCore era has come: the sooner you jump in, the better.