The State of the Art in GPU Computing

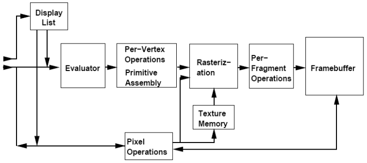
NVIDIA and SINTEF
HPC GPU Computing Mini-Workshop

Jon Mikkelsen Hjelmervik
<jon.m.hjelmervik@sintef.no>

The Heterogeneous Computing Group at SINTEF ICT Applied Mathematics

Outline

Prelude: What is a GPU?



(From the OpenGL 2.0 specification)

Evolution into the GPU:

- Basic HW accelerating rendering,
- increasingly more powerful,
- and increasingly more flexible,
- driven by computer games

	1996	1997	1998	1999	. . .	2009
Application tasks	CPU	CPU	CPU	CPU		CPU/GPU
Scene level calculations	CPU	CPU	CPU	CPU		CPU/GPU
Transformations	CPU	CPU	CPU	GPU		GPU
Lighting calculations	CPU	CPU	CPU	GPU		GPU
Clipping and triangle setup	CPU	graphics processor	graphics processor	GPU		GPU
Rasterization	graphics processor	graphics processor	graphics processor	GPU		GPU

GPUs and CPUs: Head-to-head comparison

	CPU	GPU (Fermi/G200)
Full cores	**4**	—
Accelerator cores	0	16/**30**
Arithmetic units	**16**	**512**/**240**
Intercore communication	Cache	L2 Cache/None
SIMD width	4	32/32
Float operations per cycle	16	1024/720
Frequency (GHz)	**3.2**	**?** /**1.3**
Single precision gigaflops	102	?/936
Double:single performance	1:2	1:2/1:12
Gigaflops / watt	0.8	?/5
Megaflops / USD	70	?/550
Accelerator Bandwidth (GB/s)	N/A	?/102
Main Memory Bandwidth (GB/s)	25.6	8
Maximum memory size (GiB)	24	6/4

Key observations:

- GPUs have considerably more cores than CPUs.
- GPUs run at a lower clock frequency than CPUs.

So what's the thing with all these cores?

Power density is Watts per area

- Power density indicates the heat produced:
  - Power density of CPUs has surpassed that of a cooking plate.
  - Cooling is a big problem.
- Why is the power density so high?
  - Higher clock frequencies require higher supply voltages.
- High power usage is undesirable:
  - battery life, environmental concerns, etc. . .

Power density is proportional to cube of voltage

Reduce freq & voltage by 1% reduces power density by 3%:

- **One** core at 100% freq & voltage offers 100% performance.
- **Two** cores at 85% freq & voltage offers 180% performance, and **consume approximately the same amount of power!**

GPUs and CPUs: Head-to-head comparison

	CPU	GPU (Fermi/G200)
Full cores	4	—
Accelerator cores	0	16/30
Arithmetic units	16	512/240
Intercore communication	Cache	L2 Cache/None
SIMD width	4	32/32
Float operations per cycle	16	1024/720
Frequency (GHz)	3.2	?/1.3
Single precision gigaflops	**102**	?/**936**
Double:single performance	1:2	1:2/1:12
Gigaflops / watt	**0.8**	?/**5**
Megaflops / USD	**70**	?/**550**
Accelerator Bandwidth (GB/s)	N/A	?/102
Main Memory Bandwidth (GB/s)	25.6	8
Maximum memory size (GiB)	24	6/4

- Massive parallelism (requires new methods)
- Development driven by computer games.
- Mass production: R&D cost divided by millions of units sold.

Why Now?

Power wall

- Power density cannot increase further

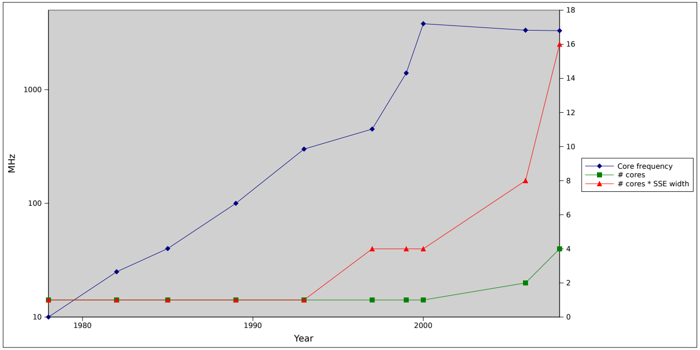
Instruction Level Parallelism wall

- Introducing further logic to improve ILP does not pay off

Memory wall

- Memory latency is getting too high

CPU development

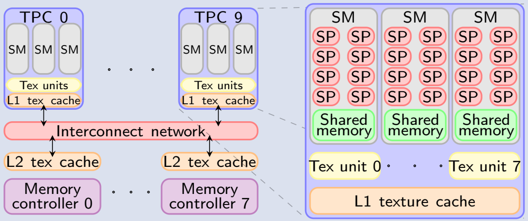


Bottom line

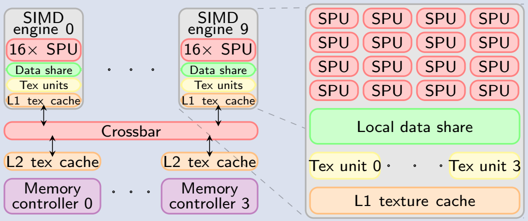
- Serial programming belongs to the past
- Most programs need modifications for upcoming processors

How are GPUs organized?

NVIDIA GT200 GPU



AMD RV770 GPU



Fermi

New features

- Floating point compliance (IEEE 754 2008)
  - More compliant than most CPUs
  - More compliant than SSE instructions
- Drastically increased double precision performance
- Support for ECC memory

Performance

Memory transfers

- Small cache
- Associated threads should access the same memory bank

Compute

- High peak floating point performance compared to bandwidth
- Divergent code hurts performance

Programming model

Host program

- Executed on the CPU
- Controls data flow to and from graphics memory
- Initiates and controls GPU programs

Kernel

- Executed in parallel on the GPU
  (typically tens of thousands or more instances)
- Performs the computation
- Implicitly invoked

Example kernel

```
__global__ void
addKernel(float* c, float* a, float* b)
{
  int x = blockIdx.x*blockDim.x+threadIdx.x;
  int y = blockIdx.y*blockDim.y+threadIdx.y;
  int w = gridDim.x*blockDim.x;
  c[y*w+x] = a[y*w+x] + b[y*w+x];
}
```

- a, b and c are pointers to global graphics memory.
- `threadIdx` contains the index of the current thread

Programming languages

	OpenCL	**CUDA**	**Brook+**
Target platform	GPU ++	GPU (NV)	GPU (AMD)
Abstraction	API	API, compiler	API, compiler
Host language	C	C, C++	C++
Kernel language	C99-based	C99-based, some C++	C99-based
Memory model	∼PGAS	∼PGAS	Data streams
Task-parallelism	Full	Streams	Streams
Ease of use	⋆⋆	⋆⋆	⋆⋆
Maturity	⋆	⋆⋆⋆	⋆⋆

- **CUDA**: designed to expose the compute capabilities of Nvidia GPUs. Supports C++ kernels. Currently, most popular choice.
- **Brook+**: designed to expose the compute capabilities of AMD GPUs. Higher abstraction level.
- **OpenCL**: Upcoming industry standard, SDKs from Nvidia and AMD released fall 2009.

Dense linear algebra

Application areas

Important building block for a wide range of applications, including simulations

Characteristics

- Highly optimized libraries exist for a wide range of architectures, including GPUs
- Practically branching free
- Linear memory access pattern

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Dense matrix-vector product

Characteristics

- Each matrix element is used only once

GPU results

- Parts of the vector can be kept in shared memory
- Bandwidth limited, making it unable to take advantage of wide parallel architectures
- The high bandwidth makes GPUs attractive choice

$$\begin{bmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & \mathbf{a_{13}} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_3} \end{bmatrix} = \begin{bmatrix} \mathbf{b_1} \\ b_2 \\ b_3 \end{bmatrix}$$

Dense matrix-matrix product

Characteristics

- No dependant data accesses
- High reuse of data

GPU results

- Blocks of the matrices can be kept in shared memory
- Near peak floating-point performance can be achieved for a wide range of architectures. (Volkov & Demmel, 2008)

$$\begin{bmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & \mathbf{a_{13}} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & \mathbf{b_{12}} & b_{13} \\ b_{21} & \mathbf{b_{22}} & b_{23} \\ b_{31} & \mathbf{b_{32}} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & \mathbf{c_{12}} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

Sparse matrix-vector multiplication

Application areas

Commonly used as part of linear solver, i.e. conjugate gradient methods

Characteristics

- Only nonzero matrix elements are stored
- Unstructured memory accesses
- Divergent control flow
- Difficult to get high performance (also for CPU implementations)
    - CPU implementations typically achieve 10% bandwidth utilization and less than 1% compute utilization (Göddeke *et al.*, 2007)
- Performance is problem dependent

Sparse matrix-vector multiplication (cont)

GPU results

One of the first problems studied for GPGPU (Krüger & Westermann, 2003)

Hybrid data format for GPU implementation

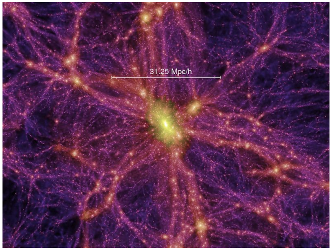
To overcome unbalanced workloads and non-coalesced memory reads, one may use hybrid data formats

- Store structured parts of the matrix in a dense format.
- Remaining elements are stored in sparse format
- Performance is related to the efficiency off the dense matrices
- Many matrices can be reorganized to improve efficiency
- $10\times$ faster than CPU implementations (Bell & Garland, 2008)

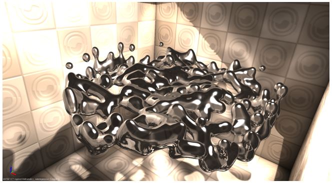
N-body

Application areas

Particle simulation, SPH, the Millennium Run etc.



(Source: wikipedia.org)



(Source: SINTEF)

N-body (cont)

Characteristics

- All particles affect each other
- Can be simplified by treating groups far away as a single contribution
- Each particle can be computed independently

GPU challenges

- Unstructured memory access
- Finding nearest neighbors

Stencil computations

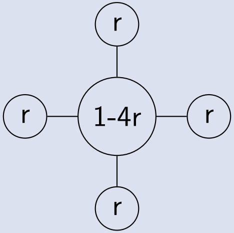
Application areas

PDE solvers and image processing

Characteristics

Weighted average of a neighborhood is computed for each cell in the grid.

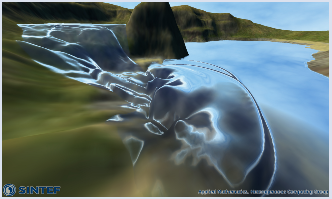
- Typically bandwidth limited



Stencil computations (cont)

GPU results

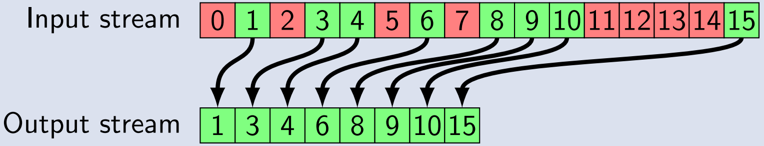
- Performance improvement by domain decomposition
- Nonlinear schemes can yield up to 30× speedup (Hagen *et al.*, 2005)



Stream compaction and expansion

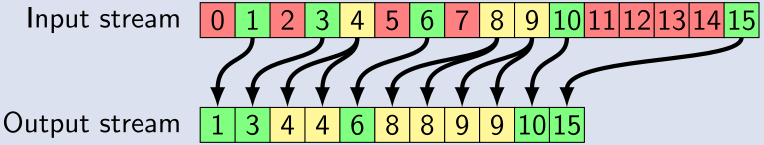
Stream compaction

- Make compact stream of arbitrarily input stream elements



Stream compaction and expansion

- Allow input stream elements repeat in output stream



Stream compaction and expansion (cont)

Application areas

Search or reorganization of data

Characteristics

- Trivial if programmed serially
- Tricky to do efficiently in parallel

GPU results

- CUDPP library (Harris *et al.*, n.d.)
  - Implements scan (Hillis & Steele Jr, 1986), (Blelloch, 1990)
  - Work-horse for a variety of algorithms
  - Requires scatter write
- Histogram Pyramids (Ziegler *et al.*, 2006; Dyken *et al.*, 2008)
  - Good for sparse extraction
  - No scatter, can be implemented in plain OpenGL.

Sorting

Application areas

Sorting is useful in a wide range of applications, including computer graphics and databases

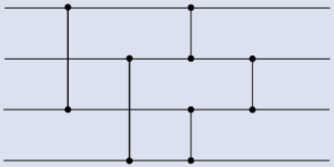
Characteristics

- Traditional approaches (quicksort) do not map to modern processors
    - Hard to parallelize (load balancing)
    - Hard to vectorize

Sorting (cont)

GPU results

- Early GPU algorithms are based on Batcher's sorting network (Purcell *et al.*, 2003)



- Current GPU algorithms is based on two-step algorithm
  - Sort a sub-vector fitting in shared memory
  - Merge the sub-vectors
- This strategy is also well suited for multi-core CPUs (Satish *et al.*, 2008)

Concluding remarks

- Wide area of applications can benefit from GPU computing
- Concepts from GPU computing is useful for multicore CPUs
- Algorithms must become data parallel
  - Serial code utilize 6% of CPU potential (4-core SSE)
  - Existing code must be redesigned and rewritten (not just a new compile target...)
  - Redesigned code tend to execute 10× faster on the GPU
- Industry adoption is considerate and growing

Thank you for your attention!

Bibliography I

Bell, N., & Garland, M. 2008 (Dec.).
Efficient sparse matrix-vector multiplication on CUDA.
NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

Blelloch, G. 1990 (Nov.).
Prefix sums and their applications.
Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Dyken, C., Ziegler, G., Theobalt, C., & Seidel, H.-P. 2008.
High-speed marching cubes using histogram pyramids.
Computer graphics forum, **27**(8), 2028–2039.

Bibliography II

Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S., Grajewski, M., & Turek, S. 2007.
Exploring weak scalability for FEM calculations on a GPU-enhanced cluster.
Parallel comput., **33**(10–11), 685–699.

Hagen, T., Hjelmervik, J., Lie, K.-A., Natvig, J., & Henriksen, M. 2005.
Visual simulation of shallow-water waves.
Simulation modelling practice and theory, **13**(8), 716–726.

Bibliography III

Harris, M., Owens, J., Sengupta, S., Zhang, Y., & Davidson, A.
CUDPP: CUDA data parallel primitives library.
http://www.gpgpu.org/developer/cudpp/.
[visited 2009-03-20].

Hillis, W., & Steele Jr, G. 1986.
Data parallel algorithms.
Commun. acm, **29**(12), 1170–1183.

Krüger, Jens, & Westermann, Rüdiger. 2003.
Linear algebra operators for gpu implementation of numerical algorithms.
Pages 908–916 of: Siggraph '03: Acm siggraph 2003 papers.
New York, NY, USA: ACM.

Bibliography IV

Purcell, T., Donner, C., Cammarano, M., Jensen, H., & Hanrahan, P. 2003.
Photon mapping on programmable graphics hardware.
Pages 41–50 of: Eurographics.
Eurographics Association.

Satish, N., Harris, M., & Garland, M. 2008 (Sept.).
Designing efficient sorting algorithms for manycore GPUs.
NVIDIA Technical Report NVR-2008-001. NVIDIA.

Volkov, V., & Demmel, J. 2008.
Benchmarking GPUs to tune dense linear algebra.
Pages 1–11 of: Supercomputing.
Piscataway, NJ, USA: IEEE Press.

Bibliography V

ZIEGLER, G., TEVS, A., THEOBALT, C., & SEIDEL, H.-P.
2006.
GPU point list generation through histogram pyramids.
Tech. rept. MPI-I-2006-4-002. Max-Planck-Institut für Informatik.