Advanced Numerical Methods on GPUs

Dominik Göddeke and Stefan Turek

Institut für Angewandte Mathematik (LS3)
TU Dortmund

dominik.goeddeke,stefan.turek@math.tu-dortmund.de
http://www.mathematik.tu-dortmund.de/~goeddeke
http://www.mathematik.tu-dortmund.de/LS3

ENUMATH 2011 Mini-Symposium
Leicester, UK, September 7

technische universität dortmund

fakultät für mathematik

The Big Picture

Problems with current hardware

- Memory wall: Data movement cost prohibitively expensive
- Power wall: Nuclear power plant for each machine (in the cloud)?
- ILP wall: 'Automagic' maximum resource utilisation?
- Memory wall + power wall + ILP wall = brick wall

Inevitable paradigm shift: Parallelism and heterogeneity

- In a single chip: singlecore $\rightarrow$ multicore, manycore, ...
- In a workstation (cluster node): NUMA, CPUs and GPUs, ...
- In a big cluster: different nodes, communication characteristics, ...

This is our problem as mathematicians

- Affects standard workstations and even laptops
- In most cases, cannot be hidden from us properly

Consequences for Numerics

Without respecting parallelism

- Impossible to exploit ever increasing peak performance
- Sequential codes even run slower on newer hardware (!)

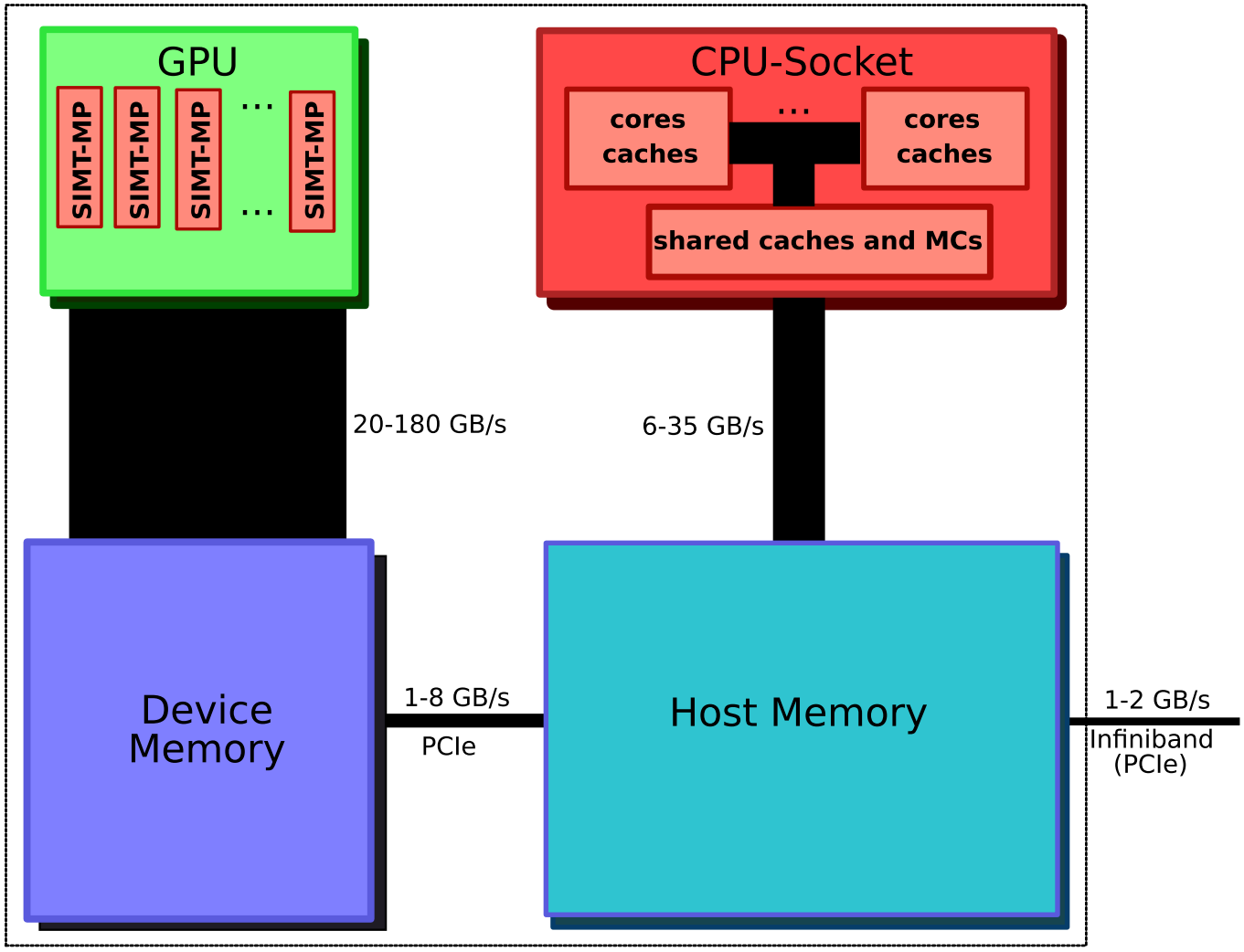
Challenges

- Technical: Compilers can't solve these problems, libraries are limited
- Numerical: Traditional methods often contrary to hardware trends
- Goal: Redesign existing numerical schemes (and invent new ones) to work well in the fine-grained parallel setting

GPUs ('manycore') are forerunners of this development

- 10 000s of simultaneously active threads
- Promises of significant speedups
- Focus of this mini-symposium

GPUs vs. CPUs


GPU
SIMT-MP
SIMT-MP
SIMT-MP
...
SIMT-MP
CPU-Socket
cores
caches
...
cores
caches
shared caches and MCs
20-180 GB/s
6-35 GB/s
Device
Memory
1-8 GB/s
PCIe
Host Memory
1-2 GB/s
Infiniband
(PCIe)

GPUs: Myth, Marketing and Reality

Raw marketing numbers

- > 2 TFLOP/s peak floating point performance
- Lots of papers claim $> 100\times$ speedup

Looking more closely

- Single or double precision? Same on both devices?
- Sequential CPU code vs. parallel GPU implementation?
- 'Standard operations' or many low-precision graphics constructs?

Reality

- GPUs are undoubtedly fast, but so are CPUs
- Quite often: CPU codes significantly less carefully tuned
- Anything between $5 - 30\times$ speedup is realistic (and worth the effort)

Mini-Symposium Schedule

This mini-symposium

- Brief introduction to GPU computing
- Discussion of advanced numerical methods on GPUs
- State-of-the-art examples covering a wide range of numerical methods and applications

Session 1 (today): Introduction and toolkits

- 11:00–11:30: Dominik Göddeke:
  Mini-symposium welcome & introduction to GPU computing
- 11:30–12:00: Dominik Göddeke:
  Mixed-precision GPU-multigrid solvers with strong smoothers and applications in CFD and CSM
- 12:00–12:30: Mike Giles:
  OP2: An open-source library for unstructured grid applications
- 12:30: open discussion (or beating the lunch queue)

Mini-Symposium Schedule

Session 2 (tomorrow): Applications in CFD and CSM

- 11:00–11:30: Martin Geier:
  EsoStripe – An aligned data-layout for efficient CFD simulations on GPUs using the Lattice Boltzmann Method
- 11:30–12:00: Allan Peter Engsig-Karup:
  On the development of a GPU-accelerated nonlinear free-surface model for coastal engineering
- 12:00–12:30: Martin Lilleeng Sætra:
  Shallow water simulations on implicitly defined global grids
- 12:30–13:00: Christian Dick:
  CUDA FE multigrid with applications in flow/solid mechanics

Mini-Symposium Schedule

Session 3 (tomorrow): Solvers and preconditioners

- 14:00–14:30: Jan-Philipp Weiß:
  Fine-grained parallel preconditioners on GPUs and beyond
- ~~14:30–15:00: Robert Strzodka:~~
  ~~*GPU bandwidth optimization of preconditioners*~~
- 14:30–15:00: Stephan Kramer:
  Parallel preconditioning strategies for decoupled indoor air flow simulation
- 15:00–15:30: Hans Knibbe:
  GPU implementation of a Krylov solver preconditioned by a shifted Laplace multigrid method for the Helmholtz equation

Introduction to GPU Computing

Programming Languages and Existing Libraries

Pseudorandomly Chosen ‘Didactical’ Examples

Programming GPUs Directly

Obviously the most general approach

- Often unavoidable when programming for performance
- Not necessarily optimal in terms of programming effort
- Main focus of the work presented in this mini-symposium
- Rationale: When developing new numerical methods, you don't want some 'arbitrary' layer of abstraction hiding things from you

Two environments

- CUDA: More mature, bigger 'ecosystem', NVIDIA only
- OpenCL: Vendor-independent, open industry standard
- Interfaces to C/C++, Fortran, Python, .NET, ...
- Important: Hardware abstraction and 'expressiveness' are identical

Compilers and Frameworks

Compilers

- PGI Accelerator Compiler: OpenMP-like code annotations for Fortran and C
- New: Ongoing work to extend/generalise OpenMP to GPUs (!)

Frameworks

- PetSc and Trilinos: GPU support in some (important) sub-packages
- HMPP, StarPU, Quark: Load-balancing in heterogeneous systems

Standard software with GPU backends

- Matlab: GPU backends for plain Matlab and some toolboxes
- And many more: Mathematica, Ansys, OpenFOAM, ...

Standard Mathematical Libraries

Fourier Transforms

- CUFFT: NVIDIA, part of the CUDA toolkit
- APPML (formerly ACML-GPU): AMD Accelerated Parallel Processing Math Libraries

Dense linear algebra

- CUBLAS: NVIDIA's basic linear algebra subprograms
- APPML (formerly ACML-GPU): AMD Accelerated Parallel Processing Math Libraries
- CULA: Third-party LAPACK, matrix decompositions and eigenvalue problems
- MAGMA and PLASMA: BLAS/LAPACK for multicore and manycore (ICL, Tennessee)

Standard Mathematical Libraries

Sparse linear algebra and solvers

- CUSPARSE: CSR-SpMV (part of the CUDA toolkit)
- CUDPP: Building blocks for some important operations (NVIDIA and UC Davis, open-source)
- CUSP: Krylov subspace methods with simple preconditioners (NVIDIA, open-source)
- Next version of CUSPARSE: ILU(k) preconditioner
- PARDISO: sparse direct solvers

My personal two cents

- Structured case ‘solved‘, unstructured case is the challenging one!
- As always in the sparse world: Little to no standardisation

GPU Programming Model

From CPUs to GPUs on one Slide

Step 1: Simplification

- Remove caches and hard-wired logic (branch prediction, . . .)

Step 2: Invest transistors into compute

- Add as many of these 'stripped-down' cores to the chip as price/performance/power budgets allow

Step 3: 'Beef up' cores by increasing SIMD width

- 16–64 functional units per core (CPUs: 2–4) execute the same instruction in each cycle, one hardware thread per ALU
- Add local shared scratchpad memory and register file

Step 4: Tidy up

- Add several memory controllers (and graphics-specific circuits)

Architectural Key Feature of GPUs

Main difference between CPUs and GPUs

- So far, this design is not spectacularly different
- CPUs are optimised for latency of an individual task
- GPUs are optimised for throughput of many similar tasks

Key design feature: Hardware scheduler switches (groups of) threads in zero time as soon as one stalls

- Reason for stalls: Off-chip memory transfers (1000+ cycles), instructions mapping to many μ-ops, ...
- Thread creation and management entirely in hardware

Leads to programming model

- Code written for one thread, SIMD-isation done by the hardware, with some parameterisation to enable mapping of threads to data

GPU Programming Model

High level view

- Data parallelism with limited synchronisation and data sharing

Key concept: Thread blocks = virtualised multiprocessors

- Note: CUDA terminology, similar in OpenCL
- Batch computations into 'thread blocks'
- Thread blocks resident in one multiprocessor
- Blocks are independent, no guarantee of execution order
- Threads per block specified by the user (32–1024), problem-specific tunable parameter
- Threads in one block may cooperate via cheap barrier synchronisation and shared memory
- Threads from different blocks may only coordinate via global memory, synchronisation only at kernel scope

GPU Programming Model

Execution: Warps = SIMD granularity

- Threads in one block are executed in 'warps' of 32, enumerated in natural order
- One instruction per warp (SIMD granularity)
- Warps are independent, no guaranteed execution order
- Scheduler switches to next available warp in case of stall (availability due to finished memory transaction, entire block reaches barrier, . . .)
- Threads in one warp may follow different execution paths ('divergence'), resolved by serialisation and thus performance penalty

Limited resources

- Register file (32 K 4-byte entries) and shared memory (16–48 kB) are partitioned among all blocks
- Rule of thumb: Ensure at least two resident blocks per multiprocessor for good throughput ('occupancy')

Memory Subsystem

Caches

- Small global L2 cache, 768 kB currently
- Tiny L1 cache per multiprocessor, 16–48 kB
- Tiny ‘texture cache’ per memory controller, optimised for 2D locality

Parallel memory system

- 6–10 partitions, round-robin assignment in small chunks of 256 kB
- Access granularity: half-warp, i.e. 16 threads access 16 values
- Hardware may ‘coalesce’ these parallel accesses into as few as one bulk memory transaction $\Rightarrow$ crucial for performance
- Requires adhering to strict rules for memory access patterns of neighbouring threads
- Avoid ‘partition camping’, i.e. data layouts which map accesses to only one partition

Memory Subsystem

Shared memory

- 16–48 kB ‘scratchpad memory’ per multiprocessor
- Can be used as a manually controlled cache
- Common use case: Stage off-chip transfers through this memory to achieve better coalescing (different threads load data than compute on it)
- Access granularity: half-warp (16 threads)
- Physically implemented as 16-bank memory, each bank services one request at a time
- ‘Bank conflicts’: Simultaneous requests map to only one bank, resulting in serialisation and thus up to 16-fold slowdown

GPU Architecture Summary

GPUs ...

- are wide-SIMD manycore architectures
- are parallel on all levels (compute and memory)
- operate in a block-threaded way

GPUs are not

- Vector architectures (rather wide-SIMD+multithread)
- Fully task-parallel (performance stems from data parallelism)
- Easy to program efficiently (getting things running is easy though)

GPUs are particularly bad at

- Pointer chasing through memory (serialisation of memory accesses)
- Codes with lots of fine-granular branches
- Codes with lots of synchronisation and huge sequential portions

Summary: This Mini-Symposium

Parallelism and heterogeneity are inevitable

- GPUs are prominent fore-runners of this trend
- Necessary development of novel numerical methods that are better suited for the hardware: hardware-oriented numerics

GPU Architecture

- Tricky at first, especially in this crash coarse
- But: Learning curve is not so steep if one is familiar with performance tuning for CPUs

Active research topic

- 'Structured' cases pretty much solved, irregular and (at first sight) inherently sequential ones are challenging
- Algorithmic research required rather than focus on implementational details