Parallel Programming

Coprocessors

Jan Schönherr
Technische Universität Berlin
School IV – Electrical Engineering and Computer Sciences
Communication and Operating Systems (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Coprocessor

- A coprocessor is an additional (usually optional) processing unit, which is fed work by the CPU.
  - Frees CPU resources for other activities
  - Usually specialized, i. e., faster, better energy efficiency
- For example
  - Intel 80x87
  - SPEs in the Cell processor
  - GPUs (also integrated GPUs)
  - Intel Xeon Phi

CPU vs. Coprocessor

- General Purpose CPU
  - Not bad in most use cases, not good in most use cases
- Specialized Coprocessor
  - Particular good in its specific use case
  - Particular bad in everything else
- Chicken/egg problem
  - CPU is designed to execute existing software faster
  - Software is written for existing CPUs

Examples

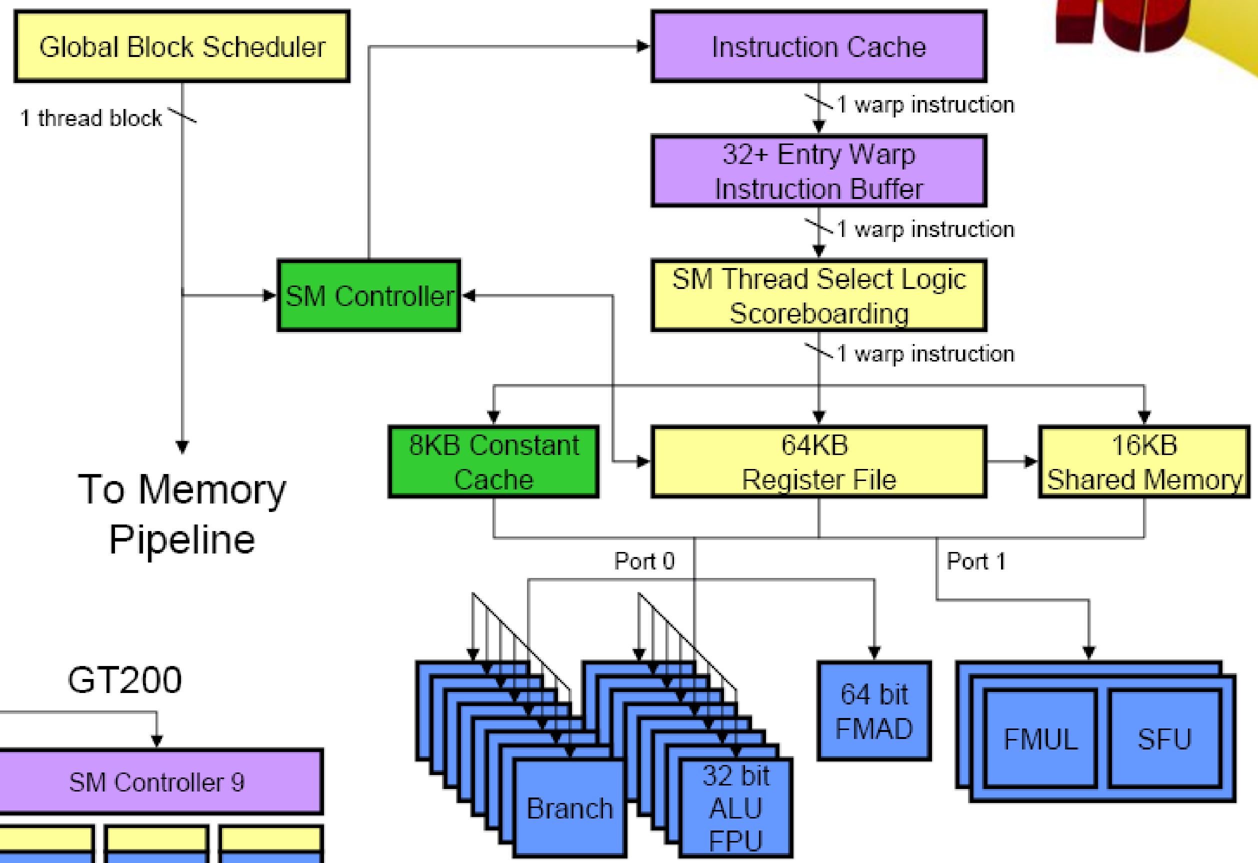
- PowerXCell 8i processor
  - 1 PowerPC Processing Element (PPE)
    - General Purpose, 2-way SMT, SIMD instructions
  - 8 Synergistic Processing Elements (SPE)
    - Single threaded, specialized instruction set
    - No direct main memory access, local storage instead (256KiB)
    - 128bit SIMD instructions
- Intel Xeon Phi
  - 60 cores, 64 bit in order x86 architecture, up to 8 GiB RAM
  - 4-way SMT (hide memory latency), 512bit SIMD instructions
  - Behaves like a “normal” system, delivered as PCIe card

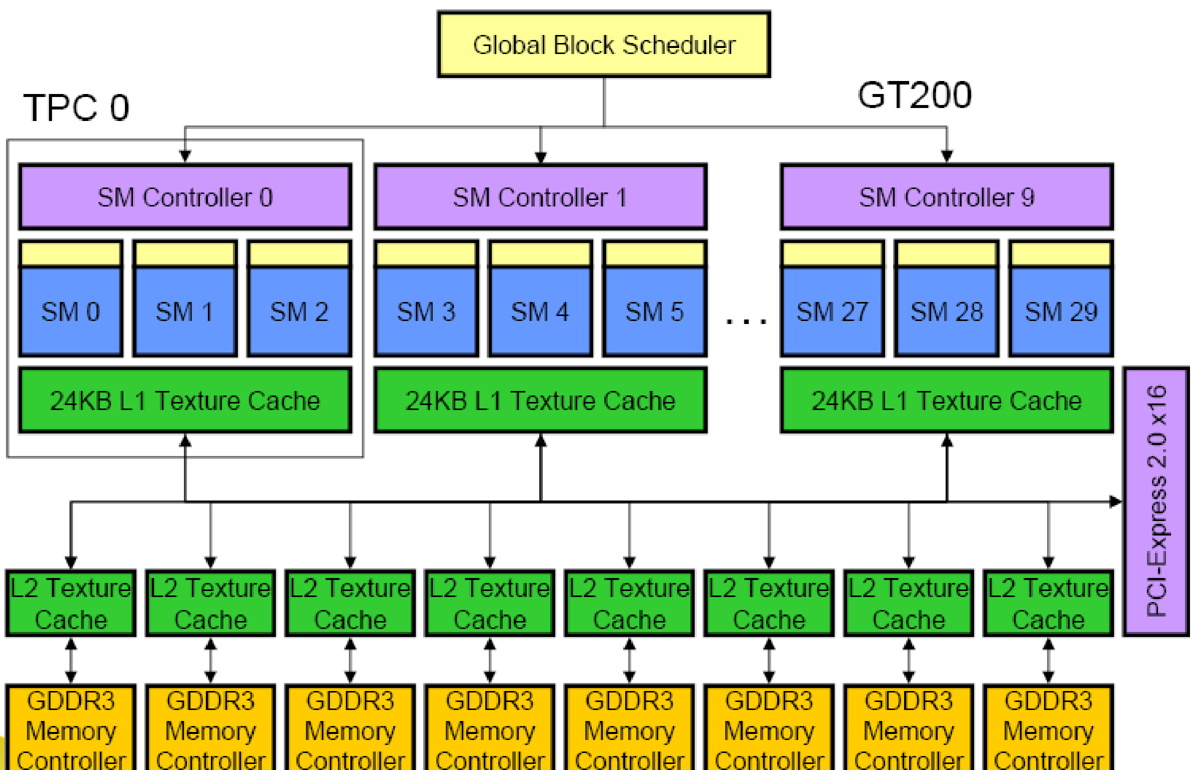
GPUs

- Optimized for SIMD-like operations
  - Many execution units, less resources for everything else
- Optimized for throughput
  - Single thread performance is not considered
  - Latencies are hidden by multi-threading

- E. g. NVIDIA GT200 family
  - Up to 30 cores (“Streaming multiprocessors”, SMs)
  - Each SM
    - Can handle up to 32 threads groups (“warps”) of 32 threads each
    - Has eight 32bit ALUs (i. e., a warp is executed in 4 steps)
    - Has only one frontend (all threads in a warp must execute the same instruction)
    - Has 16K registers, 16KiB shared memory

NVIDIA GT200





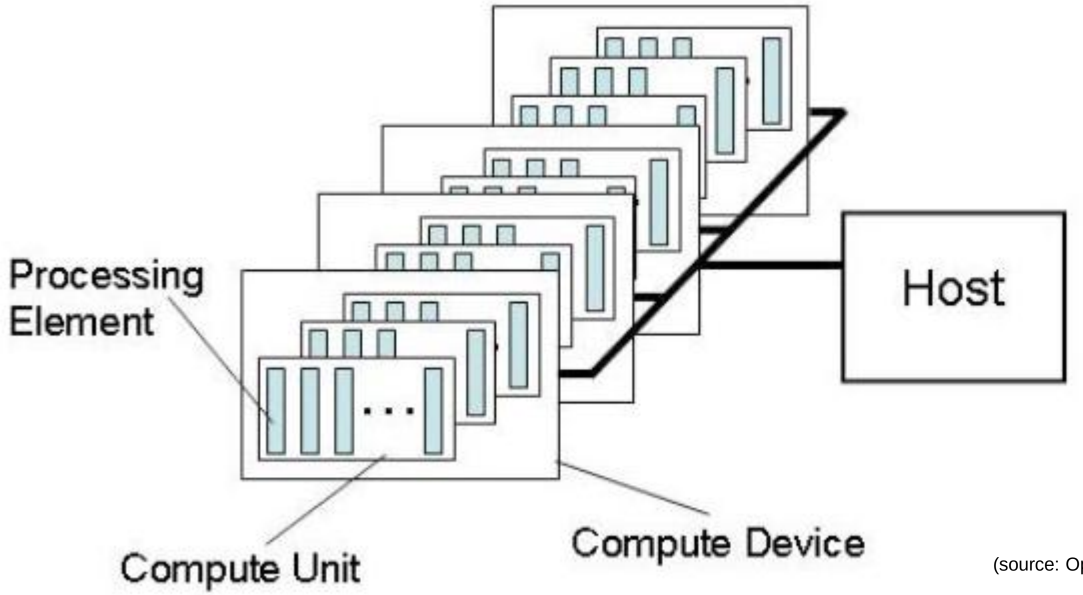
Source: realworldtech.com

General Purpose GPU Computing

- Current vendor specific APIs
  - *CUDA* for NVIDIA's GPUs
  - *ATI Stream SDK* for AMD/ATI GPUs

- Cross-vendor interface
  - OpenCL (Open Computing Language) Specification
    - Not only for GPUs
    - Tries to make coprocessor processing power accessible
      - E. g. also the "other" cores in a multi-core system
    - OpenCL 1.0 published in 2008, now at OpenCL 1.2

OpenCL – Platform Model



(source: OpenCL Specification)

- *Compute devices (CDs)* are connected to a *host*
- Compute devices consist of *compute units (CUs)* with *processing elements (PEs)*

- PEs within a CU might execute in lockstep (SIMD)
- CDs, CUs, and PEs have each individual memory
  - A PE can access its own memory, and the memory of its CU and CD
  - The host can only access the CD memories

OpenCL – Execution Model

- The *Host program* submits operations to command queues, which are processed asynchronously
  - Transfer memory to/from compute devices
  - Execute *kernels* on compute devices
  - (and synchronization commands)

- A kernel is executed on a N-dimensional index space
  - One kernel instance per index (*work-item*)
  - Index space is decomposed into homogeneous *work-groups* (which have limited size)
    - All work-items within a work-group are processed concurrently by PEs of one CU
    - Cross CU synchronization/communication is not possible!

OpenCL – Kernels

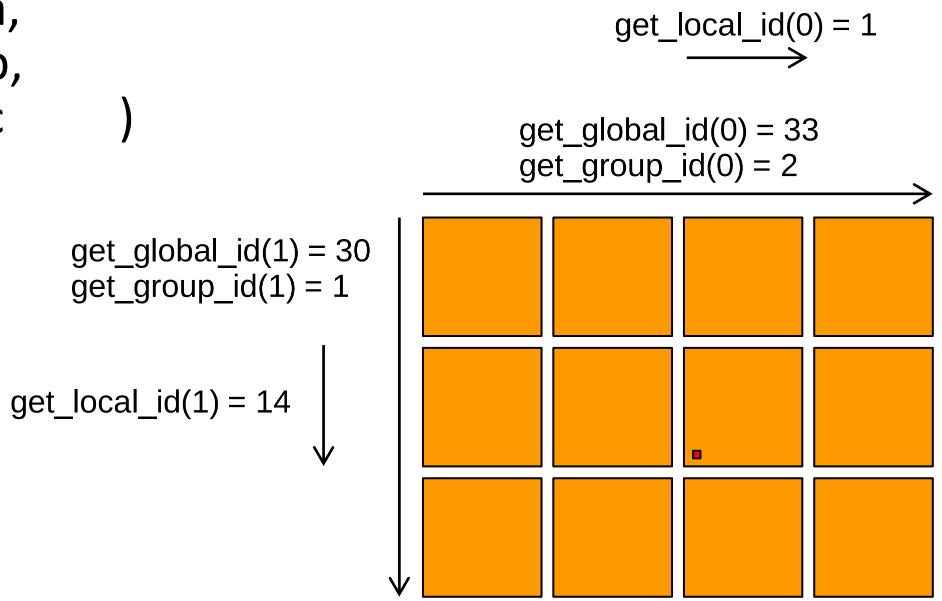
- Kernels are (usually) written in OpenCL C
  - Based on C99
  - Some extensions (e. g. vector data types)
  - Some restrictions (e. g. no recursion)

- Kernels are (usually) included in source form with the (binary) host program
  - OpenCL run-time includes a compiler
  - Compiled explicitly during execution
    - Code optimized towards the specific compute device
    - (Caching of created binary is possible)

OpenCL C

> A simple OpenCL kernel

```
__kernel void vecadd(     __global float *a,
                          __global float *b,
                          __global float *c      )
{
    int x = get_global_id(0);
    c[x] = a[x] + b[x];
}
```



get_global_id() == get_local_size() * get_group_id() + get_local_id()

> Access to different memories:
> - Global memory: __global
> - Work-group memory: __local
> - Work-item memory: __private (or without qualifier)

Considerations

- Work-items may execute in lockstep with some other work-items
  - No synchronization necessary
  - Code paths should not diverge
- Work-items within a work-group are executed concurrently
  - Synchronization possible (e. g., barrier)
  - Should take advantage of local memory
    - A variable declared __local is shared between all work-items in a work-group
- The number of work-items in a work-group is limited
  - Scalability is achieved by having many, many workgroups and a reasonable number of work-items per work-group
  - Cross work-group synchronization is not possible
    - (Only indirectly via the host; the actions of the current kernel are visible to the next kernel)

Control flow within a (simple) OpenCL host program

- Query OpenCL run-time to find a suitable compute device
  - E. g. clGetPlatformIDs(), clGetDeviceIDs()
- Setup a context with associated devices
  - E. g. clCreateContext()
- Setup other things and associate them with the context
  - Program objects: e. g. clCreateProgramWithSource(), clBuildProgram()
  - Buffer objects: e. g. clCreateBuffer()
  - Command queues: e. g. clCreateCommandQueue()
  - Kernels: e. g. clCreateKernel()
- Execute kernels
  - Copy memory between host and device: e. g. clEnqueueReadBuffer(), clEnqueueWriteBuffer()
  - Set kernel arguments: e. g. clSetKernelArg()
  - Enqueue kernel: e. g. clEnqueueNDRangeKernel()
- Free everything: clRelease*()