CUDASA: Compute Unified Device and Systems Architecture

Magnus Strengert, Christoph Müller, Carsten Dachsacher, and Thomas Ertl
Visualization Research Center (VISUS)
University of Stuttgart
Motivation

CUDA: Compute Unified Device Architecture

- Non-graphic programming interface for NVidia GPUs
- C language environment
  - Parallel programming paradigm
- Quite some interest in research community
  - approx. 30 CUDA-related publications since last year, e.g.
    - Graphic-Card Cluster for Astrophysics (GraCCA) – Performance Tests, H. Schive et al.
    - Harvesting Graphics Power for MD Simulations, J.A. van Meel et al.
    - Cuda compatible GPU as an efficient hardware accelerator for AES cryptography, S. Manavski
    - GPU Acceleration of Numerical Weather Prediction, J. Michalakes et al.
- Targeted on large, compute intense challenges
  - Often conflicts with single GPU’s hardware limitations
Motivation (cont.)

CUDA: Compute Unified Device Architecture

- Extend parallelism of CUDA for a single GPU to
  - **Bus level**: multi-GPU systems
    - SLI, Tesla, QuadroPlex, ...
  - **Network level**: GPU cluster environments
    - Myrinet, InfiniBand, Myri10G, ...
- Consistent developer interface
- Easily embedded to the CUDA compile process
System Overview

- Basic execution unit (BEU): **one single GPU** (device)
  - Device’s multiprocessors execute *kernel* blocks in parallel
  - No communication between *kernel* blocks
- Unmodified CUDA programming interface
  - No code modification required
**System Overview**

- **BEU:** **CPU + GPU** or **CPU only** (host)
  - One BEU equals one POSIX thread
  - Available BEU’s process *task* blocks in parallel
  - All blocks share common system memory (eqv. CUDA global memory)
  - Workload-balanced scheduling of blocks to the BEUs
System Overview

- **BEU: single cluster PC (node)**
  - One BEU equals one MPI process
  - MPI group process job blocks in parallel
  - No intrinsic global memory → Distributed shared memory management
  - Workload-balanced scheduling of blocks to the BEUs
System Overview

- Sequential application process
  - Arbitrary C/C++ application code
  - Allocation/Deallocation of distributed/system memory
  - Issue of function calls on network/bus level
Language Extensions

Concept:

- Unchanged CUDA syntax and semantics for GPU layer
- Minimal set of extensions for additional abstraction layers
  - Generalize present programming paradigm
  - Mimic GPU interface for all new BEUs
    - Programmability
    - Functionality (memory management, atomic functions, ...)
- Hidden underlying mechanisms for parallelism/communication
Language Extensions (cont.)

Sample CUDA code:

```c
__global__ void gFunc(float* parameter) {...} // DEVICE
__host__ void hFunc(...) {
    ...
    gFunc <<< Dg, Db, Ns >>>(parameter);
}

__task__ void tFunc(float* parameter) { // HOST
    hFunc(parameter);
}

__sequence__ void sFunc(...) { // APP
    tFunc <<< Dg >>>(parameter);
}
```

Sample CUDASA code:

Generalized Execution Configuration

New Function Qualifier

Function Qualifier
## Language Extensions (cont.)

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Exposed</th>
<th>Internal</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>application layer</td>
<td><strong>sequence</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>network layer</td>
<td><strong>job</strong></td>
<td><strong>node</strong></td>
<td>jobIdx, jobDim</td>
</tr>
<tr>
<td>bus layer</td>
<td><strong>task</strong></td>
<td><strong>host</strong></td>
<td>taskIdx, taskDim</td>
</tr>
<tr>
<td>GPU layer</td>
<td><strong>global</strong></td>
<td><strong>device</strong></td>
<td>gridDim, blockIdx, threadIdx</td>
</tr>
</tbody>
</table>

- Exposed functions are accessible from next higher abstraction
- Built-ins are automatically propagated to all underlying layers
Implementation

- **Runtime Library**
  - *Job* and *task* scheduling
  - Idle BEUs request new workload
  - Distributed shared memory for network layer
    - `cudasaMalloc`, `cudasaMemcpy`, `cudasaFree`, ...
  - Common interface functions (e.g. atomic functions)
    - `cudasaAtomicAdd`, `cudasaAtomicSub`, `cudasaAtomicExch`, ...

- **CUDASA Compiler**
  - Code translation from CUDASA code to CUDA with threads/MPI
    - Self-contained pre-compiler to CUDA compiler process
      - Based on Elsa (C++ parser) with added CUDA/CUDASA functionality
    - Full analysis of syntax and semantics required
Implementation: Bus Layer

Example code translation:

```c
__task__ void tFunc(int i, float *f) { ... }

typedef struct {
    int i; float *f;
    dim3 taskIdx, taskDim;
} wrapper_struct_tFunc;

void tFunc(wrapper_stuct_tFunc *param) {
    int i = param->i; float *f = param->f;
    dim3 taskIdx = param->taskIdx;
    dim3 taskDim = param->taskDim;
    { ... }
}
```
Example code translation:

```c
UNUSED tFunc <<< Dg >>>(i, f);
```

Generated code layout:

- Copy function parameters into wrapper struct `(i, f)`
- Populate scheduler queue with all blocks of the `taskgrid` `(Dg)`
- Determine built-ins for each block `(taskIdx, taskDim)`
- Wake up BEU worker threads from the thread pool
  - Idle BEUs request next pending block from queue
- Wait for all blocks to be processed
  - Issuing a `taskgrid` is a blocking call
Implementation: Network Layer

Code translation

- Analog to bus layer using MPI interface
- BEUs run compile-time generated event loop (eqv. to thread pool)
- Application issues broadcast messages to
  - issue execution *job functions*
  - perform shared distributed memory operations
Implementation: Network Layer

- Shared distributed memory
  - Enable computations exceeding system memory of single node
    - Each cluster node dedicates part of system memory for DSM
  - Continuous virtual address range
    - Interface via `cudasaMalloc`, `cudasaMemcpy`, `cudasaFree`
    - Evq. to CUDA global memory management
  - Implemented using MPI Remote Memory Access (RMA)
    - No guaranties for concurrent non-atomic memory accesses (as in CUDA)
    - CUDASA atomic functions for DSM
Results: Bus Parallelism

Test case 1: BLAS general matrix multiply (SGEMM)

- Build on top of CUBLAS SGEMM library function
- Block-based sub-matrix processing applied on all levels of parallelism

CPU:
- AMD Opteron 270, 2x2 cores
- Intel Q6600, 4 cores

GPU:
- NVIDIA Quadro FX5600
- NVIDIA 8800GTX Ultra
- NVIDIA 8800GT
  - 2 cards (16/16 lanes)
  - 2 cards (16/4 lanes)
  - 3 cards (16/4/4 lanes)
  - 4 cards (16/4/4/4 lanes)
Results: Bus Parallelism (cont.)

Test case 2: Sub-problem of global illumination

- *Implicit Visibility and Antiradiance for Interactive Global Illumination*
  C. Dachsbacher et al., 2007
- “Local Pass”: Convolution of incoming radiance with surface BRDF
  - Partition of all scene elements in task and kernel blocks
  - Uniform directional radiance distribution (128 samples)

<table>
<thead>
<tr>
<th>Number of scene elements</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768</td>
<td>526</td>
<td>263</td>
<td>129</td>
</tr>
<tr>
<td>131072</td>
<td>1030</td>
<td>520</td>
<td>265</td>
</tr>
</tbody>
</table>

In milliseconds for a single-node multi-GPU system with four NVIDIA 8800GTs
Results: Network Parallelism

- Proof of concept: SGEMM using network parallelism
  - Test case 1:
    - 2 cluster nodes, two 8800GTs each, Gigabit Ethernet
    - $25000^2$ matrices $\rightarrow$ **192 GFlops**
      - Job computation: 1.6s
      - Job communication: 4.9s
    $\rightarrow$ High communication costs

- Test case 2: Simulation of “optimal network”
  - Single PC with 4 GPUs used as 4 single GPU cluster nodes
  - $10240^2$ matrices $\rightarrow$ **236 GFlops** (bus level only: 314 GFlops)
    - Requires inter-process communication
    - DSM accesses take 1.5 times longer than computation
    - No awareness of data locality
  $\rightarrow$ High (unnecessary) communication overhead
Conclusion

CUDASA: Extension to CUDA for enabling bus/network parallelism

- Minimal changes to original language
  - Low programming and learning overhead
- Good scaling behavior on bus level
  - Especially for very large target computations
- Easy to integrate into the CUDA development process

- Current project state:
  - Extension for CUDASA to add awareness of data locality
    - Idea: Callback mechanism in execution configuration
    - Minimize amount of DSM data to be communicated
    - Automatically make use of asynchronous data transfer to the GPUs
  - Preparations for making CUDASA publicly available