TStream: Scaling Data-Intensive Applications on Heterogeneous Platforms with Accelerators

Accelerators and Hybrid Exascale Systems, IPDPS’12
25th May 2012, Shanghai, China.

Ana Balevic, Bart Kienhuis
University of Leiden
The Netherlands
Motivation: Acceleration of Data-Intensive Applications on Heterogeneous Platforms with GPUs

- Tremendous compute power delivered by graphics cards

Applications, e.g. bioinformatics: Big data

Architectures: multiple devices, heterogeneity

- **Heterogeneous X*CPUs + Y*GPUs Platforms**
  - **Embedded**: TI’s OMAP (ARM+special coproc), NVIDIA Tegra
  - **HPC**: Lomonosov@1.3petaflops (1554x GPU+4-core CPUs)
Parallelization Approaches

Obtaining a Parallel Program:

Explicit Parallel Programming

Semi-Automatic (Languages, Directive-Based Parallelization)

Automatic Parallelization

Transformation frameworks

POSIX Threads

OpenMP

Intel’s TBB

CUDA

OpenACC

OpenCL

CAPS/HMPP

+run-time environments

OpenMP, TBB, StarSS, StarPU

Polyhedral Model:

data parallelism

(LooPo, Pluto, PoCC, ROSE, SUIF, CHiLL)

SM memory model

DM

our research

task + pipeline parallelism

Compaan/PNgen
Polyhedral Model: Introduction

- Static Affine Nested Loop Programs (SANLPs)
  - Loop bounds, control predicates, array references – affine functions in loop indices and global parameters
  - Host spots - streaming multimedia and signal processing applications

- Polyhedral model of a SANLP can be automatically derived based on Featurier’s fundamental work on array dataflow analysis (see: PoCC, PN, Compaan)

```
int A[X][Y]
for (int x=0; x<X; x++)
  for (int y=0; y<2*x; y++)
    tmp = A[x][y]; // Statement S
```

Example SANLP code: a for-loop nest

\[
\mathcal{D}_s = \{ \mathbf{i}_s \mid \mathbf{i}_s \in \mathbb{Z}^N, \mathbf{D}_s \mathbf{i}_s \geq 0 \}
\]

\[
\mathbf{i}_s = \{ i, n, 1 \}^T
\]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & -1 & 2 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
X \\
1
\end{bmatrix} \geq 0
\]

\[
\mathcal{F}^A_S(\mathbf{i}_s) = \mathcal{F}^A_S \mathbf{i}_s
\]

- Parallelizing/optimizing transforms on the polyhedral model, then target-specific code generation (C, SystemC, VHDL, Phtreads, CUDA/OpenCL)
Polyhedral State of The Art

- State of the art polyhedral frameworks (HPC):
  - PLuTo, CHiLL:
    • Polyhedral Model -> Coarse Grain Parallelism
    • Bondhugula et al., “PLuTo: a practical and fully automatic polyhedral program optimization system,” (PLDI’08)
    • Baskaran et al, “Automatic C-to-CUDA code generation for affine programs”, (CC’09)

- Single device (CPU or GPU), shared memory model

- Assumptions - working data set:
  - (1) resides in device memory
  - (2) always fits in device memory

  » Offloading?
  » Big data?
  » Efficient Communication?
Solution Approach

- Extension of polyhedral parallelization – compiler techniques for data partitioning into I/O tiles
- Staging I/O tiles for transfers by asynchronous entities, e.g. helper threads
- Buffered communication and streaming to GPU
Tiling + Streaming = TStream

- Stage I: Compiler transforms for data partitioning
  - Tiling in polyhedral model
  - I/O tile bounds + footprint computation
- Stage II: Support for tile streaming
  - Communication/execution mapping + tile staging
  - Efficient stream buffer design
I/O Tiling 1/2

- Tiling / multi-dimensional strip-mining
  - Decompose outer loop nest(s) into two loops
    - Tile-loop
    - Point-loop
  - Interchange

```c
int A[N], B[N], C[N];
for (int i=0; i<N; i++)
  operator(A[i], B[i], &C[i]);
```

Multi-dimensional iteration domain (here: 2-dim index vector w. supernode iterators)
Tile domain – extension of Ds with additional conditions:
\[ it \cdot TS \leq i \leq (it + 1) \cdot TS - 1 \]

```c
#define N 1000000000
#define TS (1000000)
int A[N], B[N], C[N];
for (int it=0; it<N; it+=TS)
  for (int i=it*TS; i<(it+1)*TS && i<N; i++)
    operator(A[i], B[i], &C[i]); //Statement S
```

- Coarse-grain parallelism, e.g. outer loop -> omp parallel for
- I/O Tiling – 1\textsuperscript{st} top-level tiling: Partitioning of the computation domain & Splitting working data set into smaller blocks
I/O Tiling 2/2

- **Conditions for GPU Execution**
  - All data elements must fit into the memory of the accelerator
  - Host-accelerator transfer management

- **Working data set computation**

- **I/O Tiling repeated until tile footprint is small enough to fit into GPU memory**
Tile Footprint Example

for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ )

Stencil Code: \( A_{i,j} = aA_{i-1,j} + bA_{i+1,j} + cA_{i,j-1} + dA_{i,j+1} \)
TStream:

- Stage I: Transforms for data splitting
  - Tiling in polyhedral model
  - I/O tile bounds + footprint computation
- Stage II: Support for tile streaming
  - Mapping for execution, tile staging
  - Efficient stream buffer design
Platform Mapping

- Asynchronous producer-transformer-consumer processes, implemented by helper threads executing on CPU and GPU.

- Transformer process (GPU) executes (automatically) parallelized version of computation domain, e.g. CUDA/OpenCL on GPU.

- Producer (CPU) and consumer (CPU) processes stage I/O tile DMA transfers: tile "lifting" + placement onto bus/buff.

```c
// device code executed by TS threads in parallel
__global__ computeKernel(T* A, T* B, T* C) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    operator(A[i], B[i], &C[i]); // S
}
```

Tiled Input Code. Tile Size TS = 1 Million El.

![Diagram showing process states and DMA transfers](image-url)
Efficient Stream Buffer Design for Heterogeneous Producer/Consumer Pairs

Stream Buffer
- Circular buffer w. double buffering
- Pinned host + device memory
- CUDA Streams + events combined with CPU-side sync. mechanisms

b) CPU Producer Thread
for (fid=0; fid<N; fid++){
    //push token in QA
    wait(buffQA->emptySlots);
    //produce/load
token[fid]= ...
    buffQA->put(token[fid]);
}

c) GPU Transformer Thread
for (fid = 0; fid < N; fid++) {
    //pop token from QA
    wait(buffQA->fullSlots);
    wait(buffQC->emptySlots);
    inTokenQA = buffQA->getRdPtr();
    outTokenQC = buffQC->getWrPtr();
    transformerKernel<<<NB, NT, NM, computeStream>>> (inTokenQA, outTokenQC);
    buffQA->incRdPtr();
    buffQC->incWrPtr();
    signal(buffQA->emptySlots);
    //init token push in QC
    buffQC->put(token[fid]);
}

d) Stream Buffer (FIFO)

CPU-P ➔ GPU-T ➔ CPU-C

h_data
async mem transf. 
d_data
memcpyH2D
rdptr
wrptr

host mem (pinned)
buffQA
device mem (GPU GM)

CPU
GPU

Stream Buffer
- Circular buffer w. double buffering
- Pinned host + device memory
- CUDA Streams + events combined with CPU-side sync. mechanisms
Preliminary Results

- Proof of concept: POSIX Threads + CUDA 4.0 (streams)
- Experimental Setup
  - AMD Phenom II X49653.4GHz CPU
  - ASUS M4A785TD- VEVO MB, PCIExpress 2.0 x16
  - Tesla C2050GPU (2-way DMA overlap)
- Microbenchmarks

<table>
<thead>
<tr>
<th>PCIe 2.0 x16 Mode</th>
<th>memcp yH2D</th>
<th>memcp yD2H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Directional</td>
<td>4.52-5.14 GB/s</td>
<td>5.8-6.10 GB/s</td>
</tr>
<tr>
<td>Bi-Directional</td>
<td>3.52-3.7 GB/s</td>
<td>4.34-5.39 GB/s</td>
</tr>
<tr>
<td>Bi-Directional (Concurrent)</td>
<td>2.8-3.2 GB/s</td>
<td>3.0-3.4 GB/s</td>
</tr>
</tbody>
</table>

![Graph1](chart1.png) ![Graph2](chart2.png)
Preliminary Results – Data Patterns

Vop (1:1, aligned)

Vadd (2:1, aligned)

Sobel (1*:1, non-aligned)
Conclusions

- TStream – a two phase approach for scaling data intensive applications
  - Compile-time transforms
    - I/O Tiling - Stand-alone or additional level of tiling in existing polyhedral frameworks
    - Mapping of tile access and communication code
  - Run-time support:
    - Tile streaming model - Asynchronous execution and efficient stream buffer design

- Large data processing on accelerators feasible from polyhedral model

- Enables overlapping of host-accelerator communication and computation

- First results promising, future work: integration with polyhedral process network model and the Compaan compiler framework, application studies, multi-GPU support

- Thanks to Compaan Design and NVIDIA for their support!