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

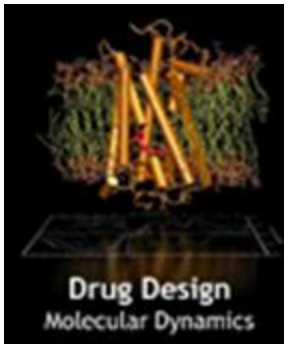
Accelerators and Hybrid Exascale Systems, IPDPS'12
25th May 2012, Shanghai, China.

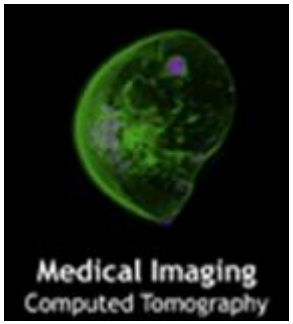
Ana Balevic, Bart Kienhuis
University of Leiden
The Netherlands

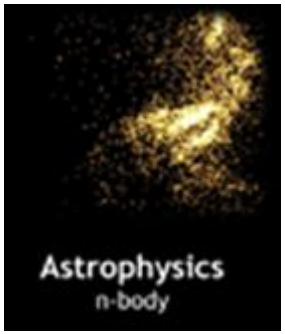
Universiteit Leiden
The Netherlands

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:

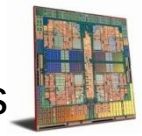
Explicit Parallel Programming

Semi-Automatic (Languages, Directive-Based Parallelization)

Automatic Parallelization

Transformation frameworks

POSIX Threads



OpenMP

Intel's TBB

Classical Compiler Analysis:
data parallelism – CETUS, PGI

CUDA

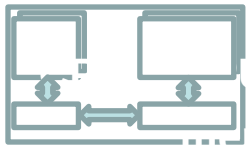
OpenACC

Polyhedral Model:

SM

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

OpenCL



CAPS/HMPP

memory model

DM

our research

+run-time environments
OpenMP, TBB, StarSS, StarPU

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

$$\mathcal{D}_{\mathbf{S}} = \{\mathbf{i_S} \mid \mathbf{i_S} \in \mathbb{Z}^N,\ \mathbf{D_S i_S} \geq \mathbf{0}\}$$

$$\mathbf{i_S} = \{\mathbf{i}, \mathbf{n}, 1\}^T$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 2 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ X \\ 1 \end{bmatrix} \geq \mathbf{0}$$

$$\mathcal{F}_S^A(\mathbf{i_S}) = F_S^A \mathbf{i_S}$$

- 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

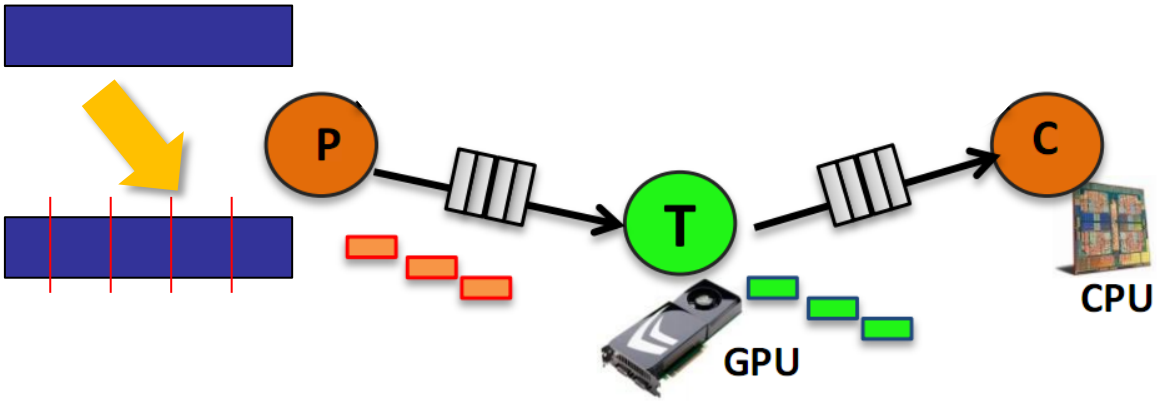
» Offloading?
» Big data?
» Efficient Communication?

Solution Approach

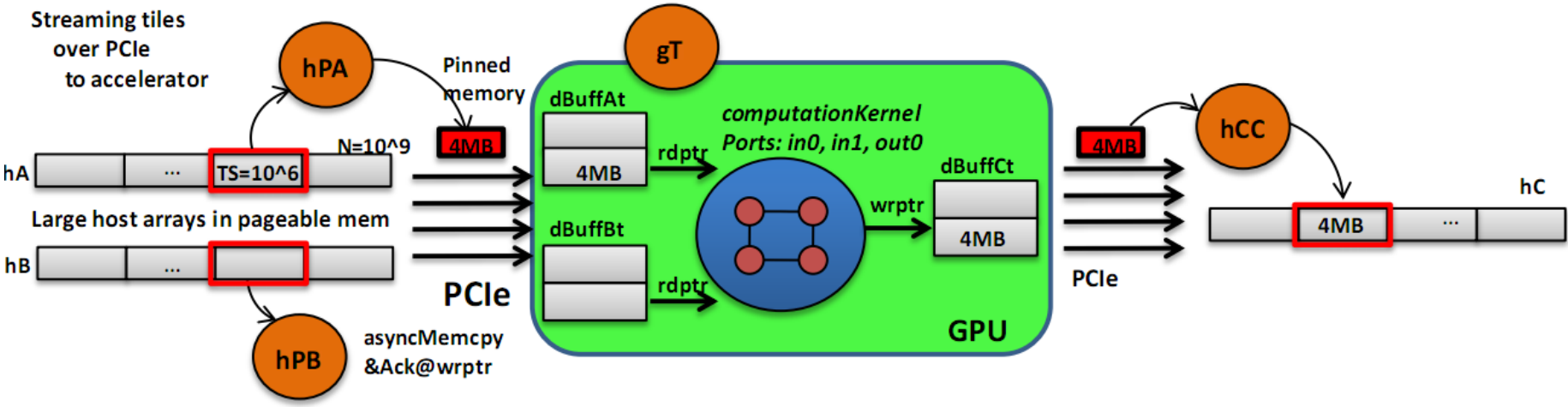
- Extension of polyhedral parallelization – compiler techniques for data partitioning into I/O tiles

- 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

```
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:

$$it \cdot TS \leq i \leq (it+1) \cdot TS - 1$$

```
#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. outter loop -> omp parallel for

- I/O Tiling – 1st 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

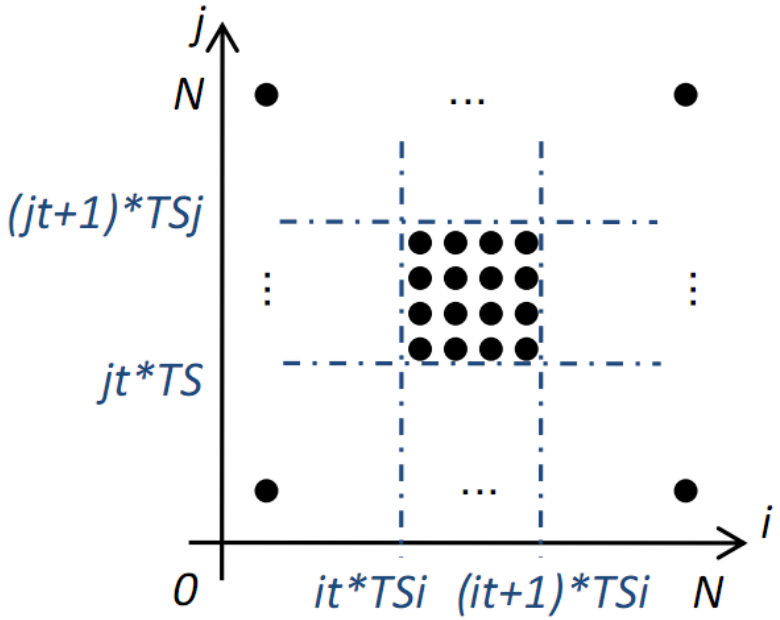
Algorithm 1 I/O Tiling for Accelerators

1: Input: Static Affine Loop Nest Program (SANLP)
2: **forall loop nests** k **in SANLP**
3: $\mathcal{D}_{\mathbf{S}} \leftarrow$ Polyhedral Domain Specification
4: Set Tile Size $ts = (s_0, s_1, .., s_{ntd})$
5: Top-Level Iteration Domain Tiling with ts Tiles
6: in: $\mathcal{D}_{\mathbf{S}}$ - constraint matrix, N-dim index vector
7: out: $\mathcal{D}^{it}_{\mathbf{S}}$ - extended constraint matrix, $(N+1)$-dim index vector
8: **for** each tile domain $\mathcal{D}^{it}_{\mathbf{S}}$ **do**
9: **for** each array A accessed by tile it **do**
10: **for** each access reference x to array A **do**
11: $F^A_S \leftarrow$ Affine Access Matrix for (A, x)
12: $\mathcal{F}^A_{S,it=k}(\mathbf{i's}) = F^A_S \mathbf{i's}$
13: $DT_{S,it=k} \leftarrow ConvexHull(\mathcal{F}^A_{S,it=k}(\mathbf{i's}))$
14: IOTile$(it, A) \leftarrow \bigcup_x$ DT(A, it, x)
15: **end for**
16: **end for**
17: TileFootprint$(it) \leftarrow \bigcup_A$ IOTile(it, A)
18: **end for**
19: **end for**

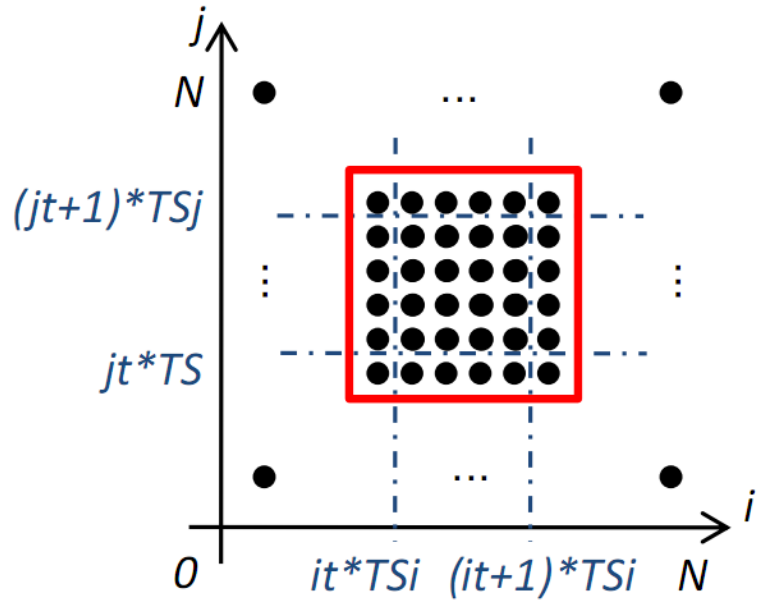
Tile Footprint Example

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

Stencil Code: $A_{i,j} = aA_{i-1,j} + bA_{i+1,j} + cA_{i,j-1} + dA_{i,j+1}$



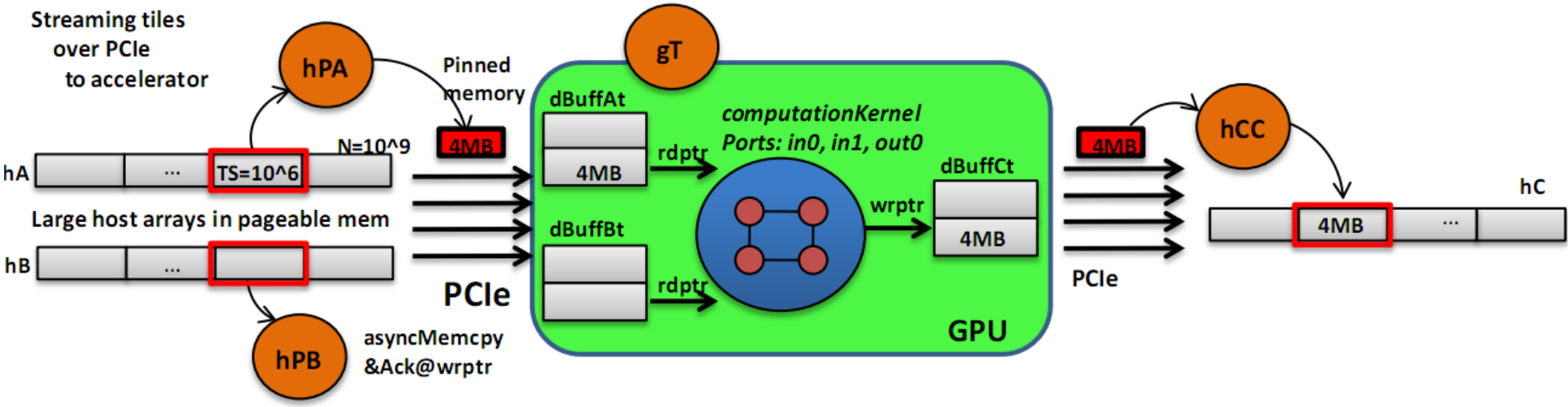
(a) Computational Tile



(b) Data Tile

TStream:

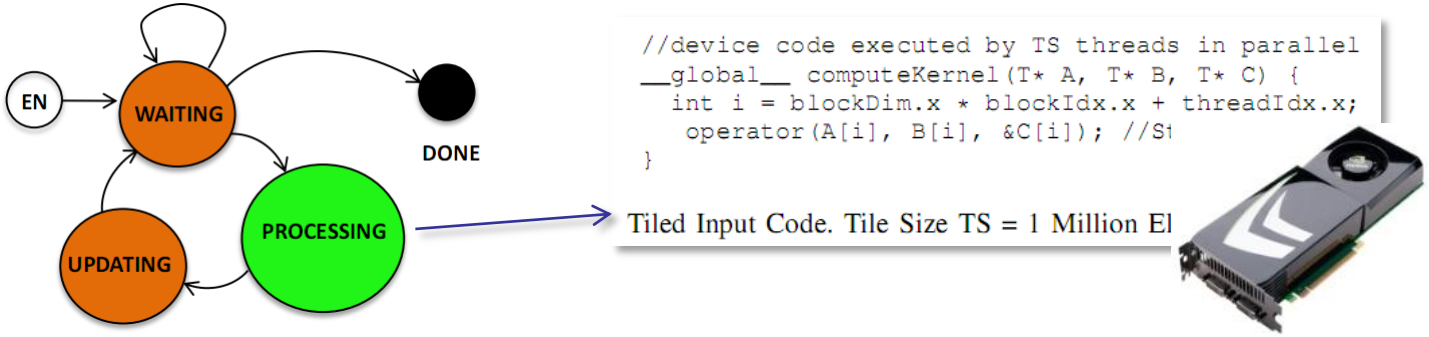
- Stage I: Transforms for data splitting
  - Tiling in polyhedral model
  - I/O tile bounds + footprint computation
- **Stage II: Support for tile streaming**
  - Mapping for execution, tile staging
  - Efficient stream buffer design



Platform Mapping

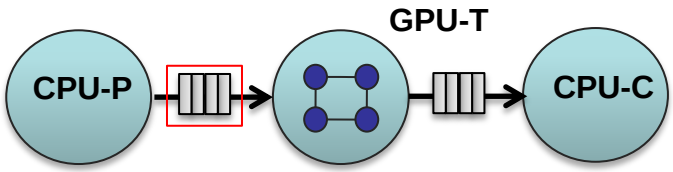
- Asynchronous producer-transformer- consumer processes, implemented by helper threads executing on CPU and GPU

- 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

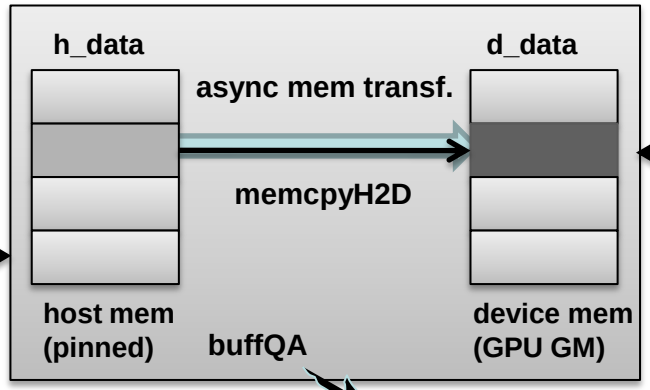
Efficient Stream Buffer Design for Heterogeneous Producer/Consumer Pairs



b) CPU Producer Thread

```
for (fid=0; fid<N; fid++){
  //push token in QA
  wait(buffQA-
>emptySlots);

  //produce/load
token[fid]
  token[fid]= …

  buffQA->put(token[fid]);
}
```

d) Stream Buffer (FIFO)



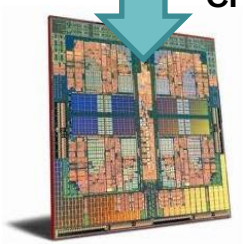
c) GPU Transformer Thread

```
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]);
}
```

e) AsyncQHandler

```
waitAsyncWriteToComplete(…);
signal(buff->fullSlots);
```

CPU



GPU

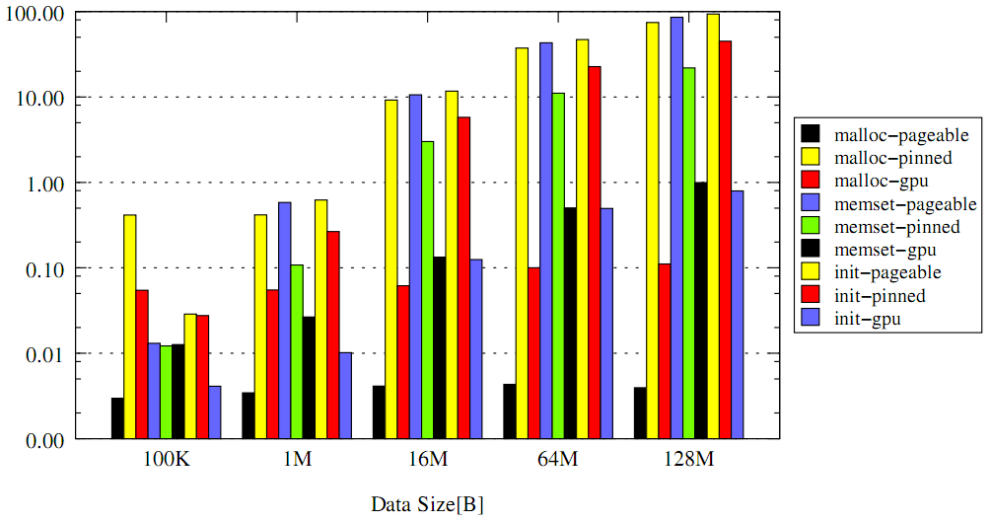
Stream Buffer

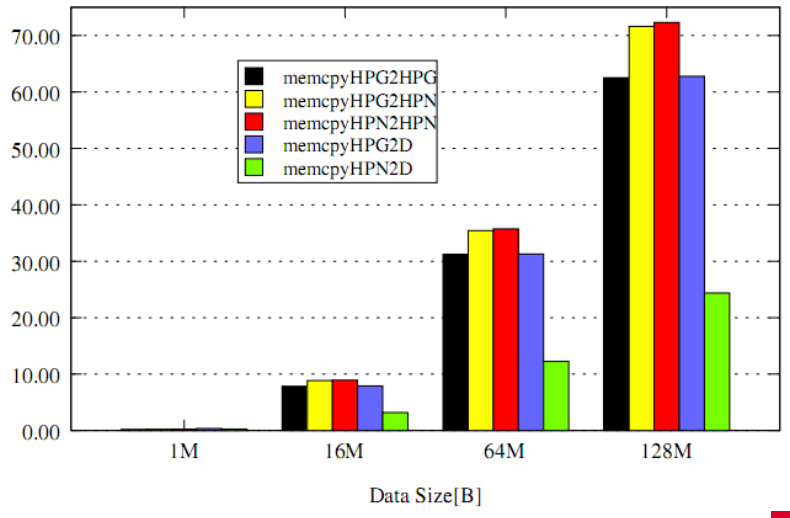
- Circular buffer w. double buffering
- Pinned host + device memory
- CUDA Streams + events combined with CPU-side sync. mechanisms

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

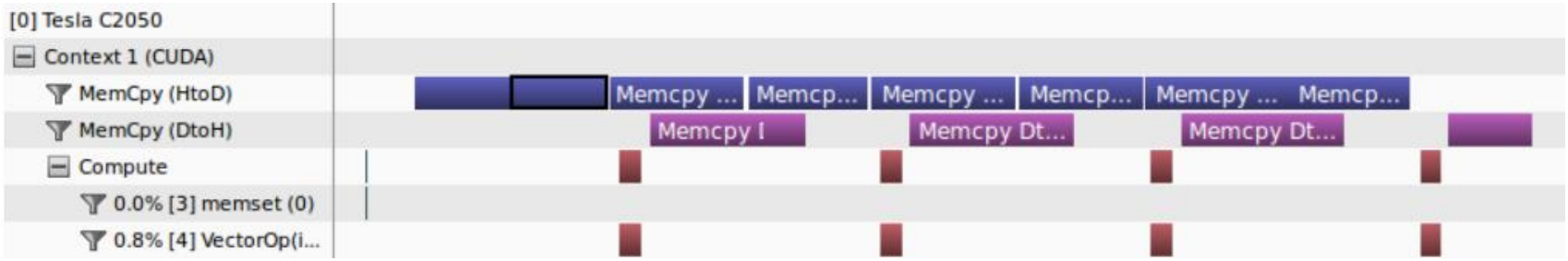
PCIe 2.0 x16 Mode	memcpyH2D	memcpyD2H
Single-Directional	4.52-5.14 GB/s	5.8-6.10 GB/s
Bi-Directional	3.52-3.7 GB/s	4.34-5.39 GB/s
Bi-Directional (Concurrent)	2.8-3.2 GB/s	3.0-3.4 GB/s



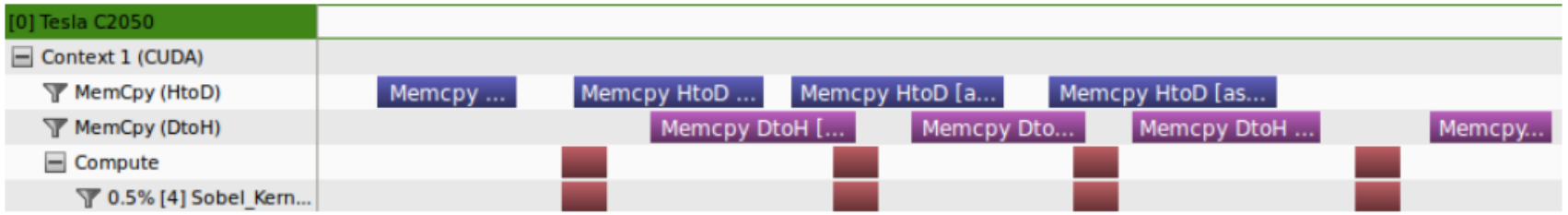


Preliminary Results – Data Patterns

Vop (1:1, aligned)



Vadd (2:1, aligned)



Sobel (1:1, non-aligned)*

NVVP

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!