Scaling Large Data Computations on Multi-GPU Accelerators

Amit Sabne, Putt Sakdhnagool,
Rudolf Eigenmann

School of ECE, Purdue University

PURDUE
UNIVERSITY

Outline

- Motivation
- Computation Splitting (COSP)
- Pipelining CPU-GPU copies
- Multi-GPU Code Generation
- Adaptive Runtime Tuning
- Evaluation
- Conclusion

PURDUE UNIVERSITY

Motivation

- GPGPUs – Accelerators of choice
- Many supercomputers use them
- Ideal for large parallel computations

However, large computations involve

- Large data sets → GPU device memories are smaller, lack good virtualization support
- Data has to be copied in and out of the GPU card → Memory copy overhead

Programmability and performance tuning have remained to be a major issues in GPGPU computing

Goals

- Automatically handle out-of-card computations
- Design a pipelining system that overlaps the kernel computations with data communications
- Implement above techniques on top of OpenMPC (OpenMP to CUDA translator)
- Automatically port OpenMP codes to multi-GPUs attached to a node
- Provide an online-tuning mechanism to choose the performance-optimal pipeline size

PURDUE UNIVERSITY

GPU Device Memory Requirement

What factors impact the GPU device memory requirement and how?

- ***Shared*** data : a single storage is required
- ***Private*** data : storage is required ***per thread*** → GPU grid size makes an impact

Some optimizations increase the memory requirement

- Prefetching (Early copy-in and late copy-out)
- Pipelining

PURDUE UNIVERSITY

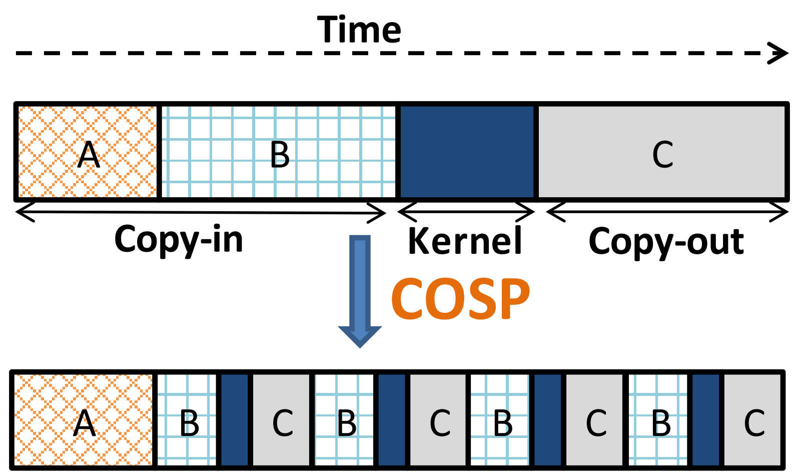
Computation Splitting (COSP)

- ***Split a large problem into smaller sub-problems*** → memory requirement reduced
- Are there side-effects?

  YES, splitting in not always perfect

  – Segregate data-types

- Data required by **every** sub-problem : ***MemFused***
- Data required **only** by a sub-problem : ***MemSplittable***



*B, C are **MemSplittable**, A is **MemFused***

PURDUE
UNIVERSITY

COSP - Code Example – Scalar Product

```
#pragma omp parallel for shared(D,E,F) private(vec, pos, sum)
for (vec = 0; vec < NUM_VECTORS; vec++) {
  sum = 0;
  for(pos = 0; pos < NUM_ELEMENTS; pos++) {
   sum += D[NUM_ELEMENTS*vec + pos]* E[NUM_ELEMENTS*vec + pos];
  }
  F[vec] = sum;
}
```

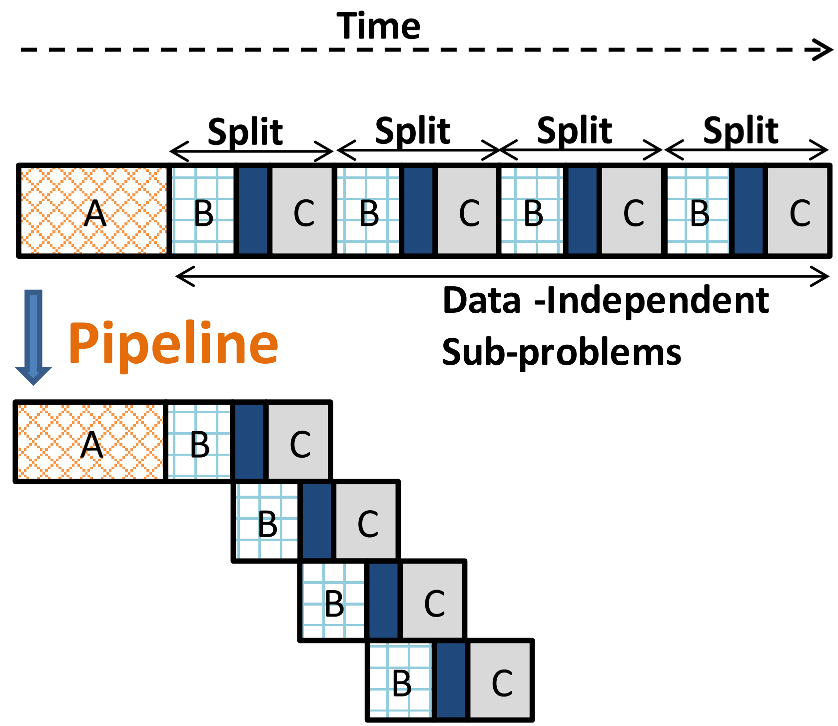
```
for (split = 0; split < NUM_VECTORS/SplitSize; split++
#pragma omp parallel for shared(D, E, F)
    private(vec, pos, sum) shared(split, SplitSize)
for (vec = 0; vec < SplitSize; vec++) {
  sum = 0;
  for(pos = 0; pos < NUM_ELEMENTS; pos++) {
    sum += D[(NUM_ELEMENTS *(vec + split*SplitSize))+ pos]
    * E[(NUM_ELEMENTS * (vec + split*SplitSize))+ pos];
  }
  F[(vec + split*SplitSize)] = sum;
}
```

Split Loop

Split

Pipelining

- Computation Splitting creates pipelining opportunities
- Resources to pipeline :
  - Copy-in channel
  - GPU cores
  - Copy-out channel
- Maximum speedup – 3*



* Considering different copy-in and copy-out channels (engines)

Pipelining – Achievable Speedup

- Computation time : $t_{compute}$
- Time required to copy ***MemFused*** data : $t_{memFused}$ (in and out of the GPU)
- For ***MemSplittable*** data,
  - t_{ci} (data copy-in time)
  - t_{co} (data copy-out time)
- Achievable Speedup

$$\frac{\text{Original Execution Time} \longrightarrow \quad t_{memFused} + t_{ci} + t_{co} + t_{compute}}{\text{Pipelined Execution Time} \longrightarrow \quad t_{memFused} + \mathbf{max}(t_{ci}\ ,\ t_{co}\ ,\ t_{compute}\ )}$$

PURDUE UNIVERSITY

Pipelining - Implementation

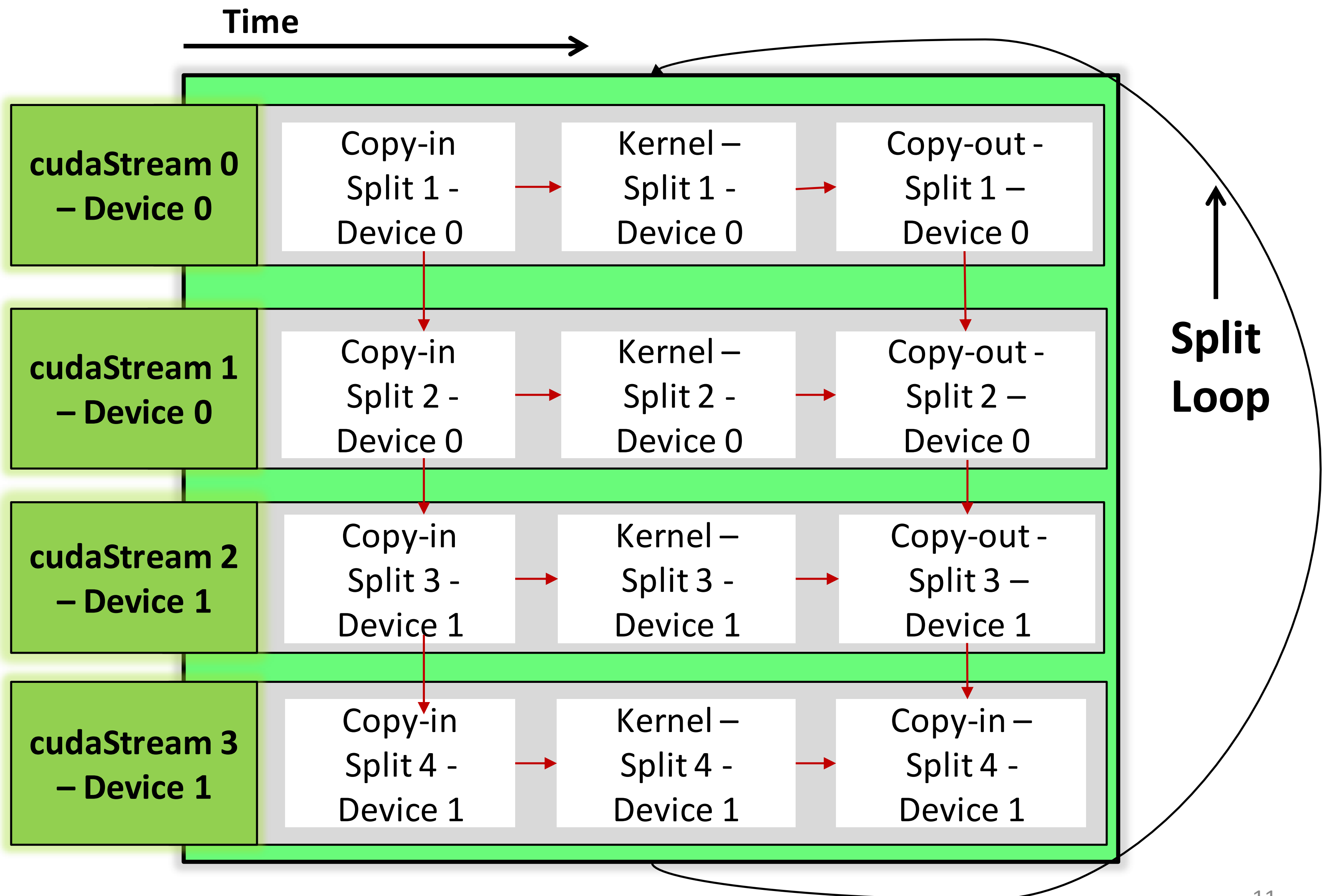
- *cudaStream* : A CUDA abstraction of instruction queues
- *cudaStreams* act independently
- Memory copy requests across *cudaStreams* get serialized
- Kernel executions across *cudaStreams* overlap

For pipelining :

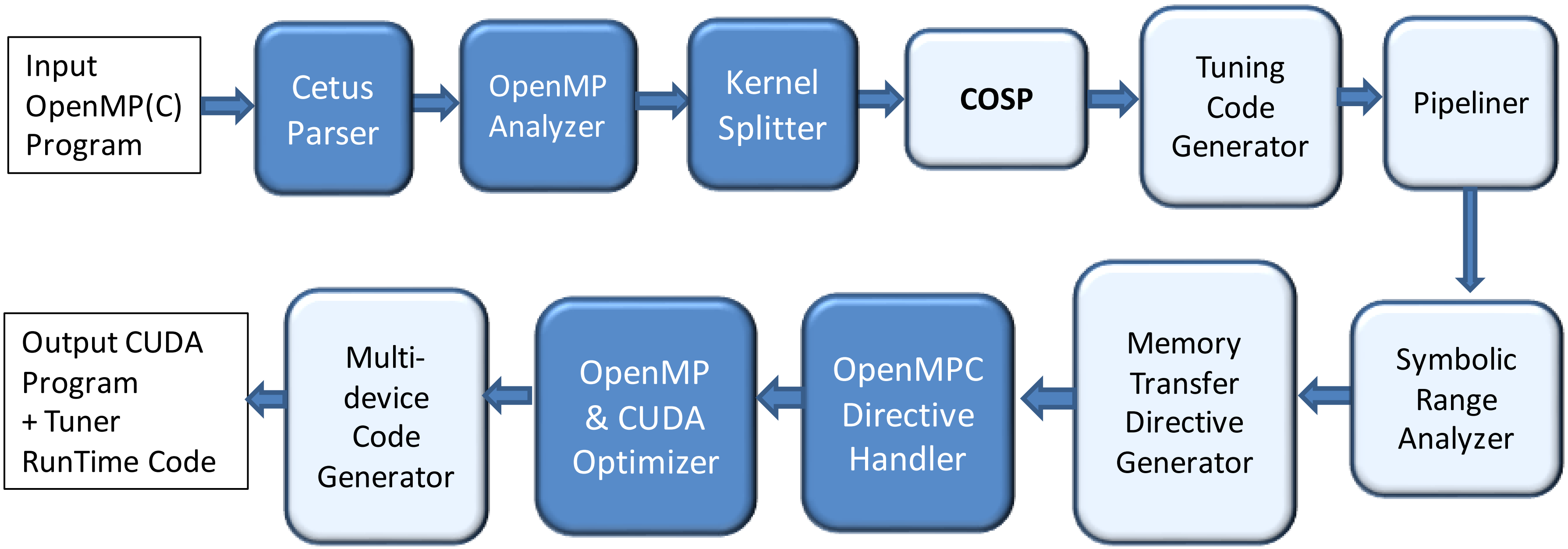
- Use 2 *cudaStreams*
- Create 2 device buffers per *MemPrivate* Data
- Create single device buffer per *MemShared* Data
- Place memory copy operations for the *MemShared* Data out of the ***Split Loop***
- Schedule alternate ***Splits*** on each *cudaStream*

PURDUE UNIVERSITY

Extending to Two GPUs



Compiler Structure



* Darker boxes are inherited from OpenMPC

PURDUE UNIVERSITY

Tuning Objective

Pipelining benefits   Equality of pipeline stage sizes

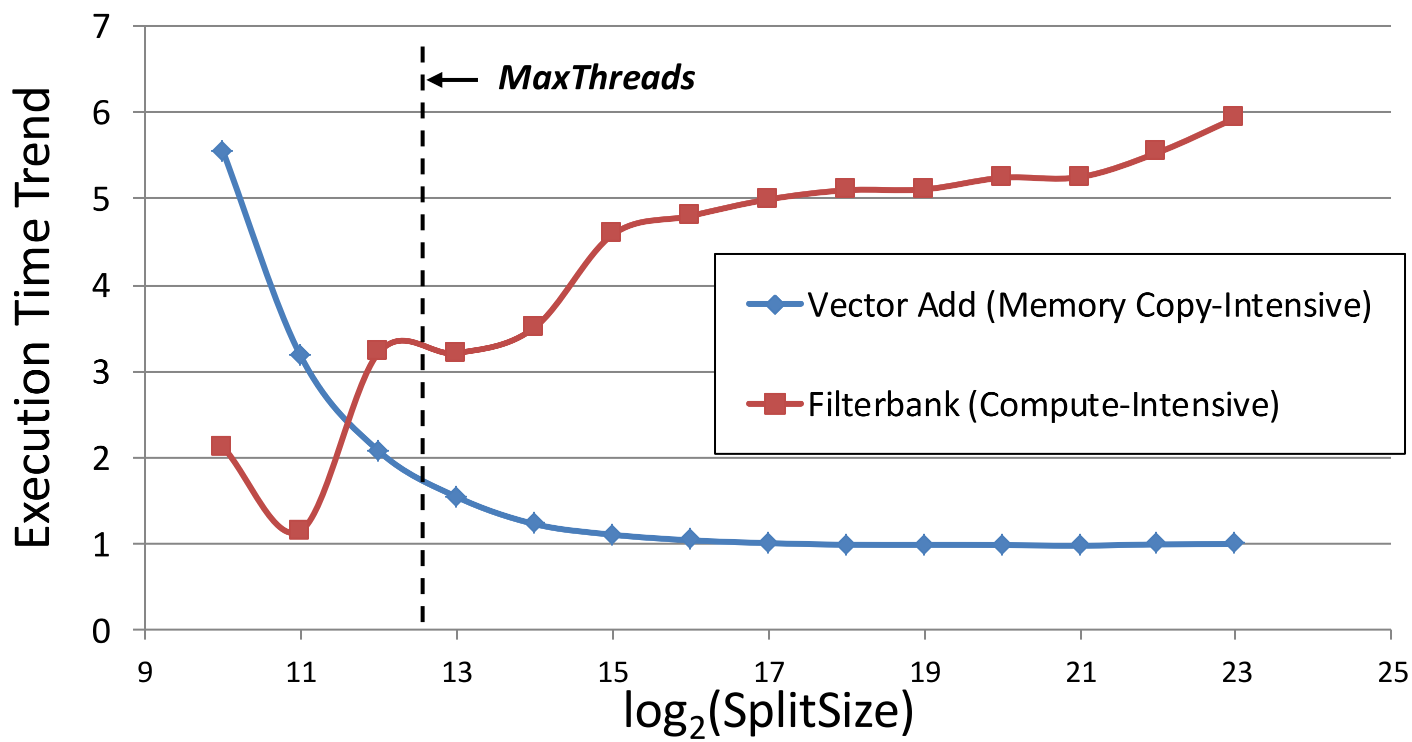
However, static scheme to determine best stage size is hard since :

- GPU systems : Intricate architecture (PCI version, #cores, GPU memory BW)
- Kernel overlaps → Difficult to model

*We propose an adaptive runtime tuning system to choose the optimal **SplitSize***

PURDUE UNIVERSITY

Compute Intensive Vs. Memory-Copy Intensive



For Compute Intensive programs, optimal SplitSize ≤ MaxThreads

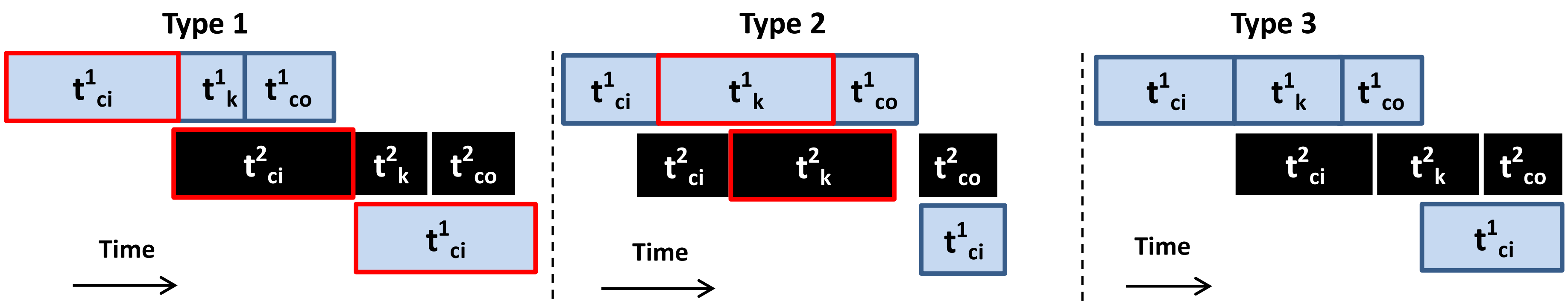
For Memory Copy-Intensive programs, optimal SplitSize ≥ MaxThreads

MaxThreads = No. of threads that can simultaneously coexist on a GPU

Heuristic Algorithm

t_{ci} = Copy-in Time t_{co} = Copy-out Time t_k = Kernel Time

(Number in superscript is the *cudaStream* number)



$t_{ci} > t_k + t_{co}$ OR $t_{co} > t_k + t_{ci}$ — **Highly Memory Copy-Intensive**

$t_k > t_{ci} + t_{co}$ — **Highly Compute-Intensive**

Other Cases

On a single Split,

- Determine the Type of the kernel
- For Type 1 kernels : Larger SplitSizes work better due to the higher bandwidth usage.
- For Type 2 kernels : SplitSizes smaller than MaxThreads are the candidates
- Generate a set of candidate SplitSizes, run each to find the best
- For Type 3 kernels : Candidate set is much larger. Type 3 is uncommon.

Tuning requires extra runs, but only on a single ***Split*** → *Tuning overhead is negligible*

Evaluation

Setup

- GPU - Tesla M2090 GPUs (4), 6GB memory, x16 PCIe link
- CPU – AMD Opteron Processor 6282, 16 cores, 2.6 GHz, 64 GB RAM

Benchmarks

- Kernels – Black Scholes, Monte Carlo, DCT, Filterbank, Vector Add, Scalar Product, FFT
- Applications – CFD, SRAD

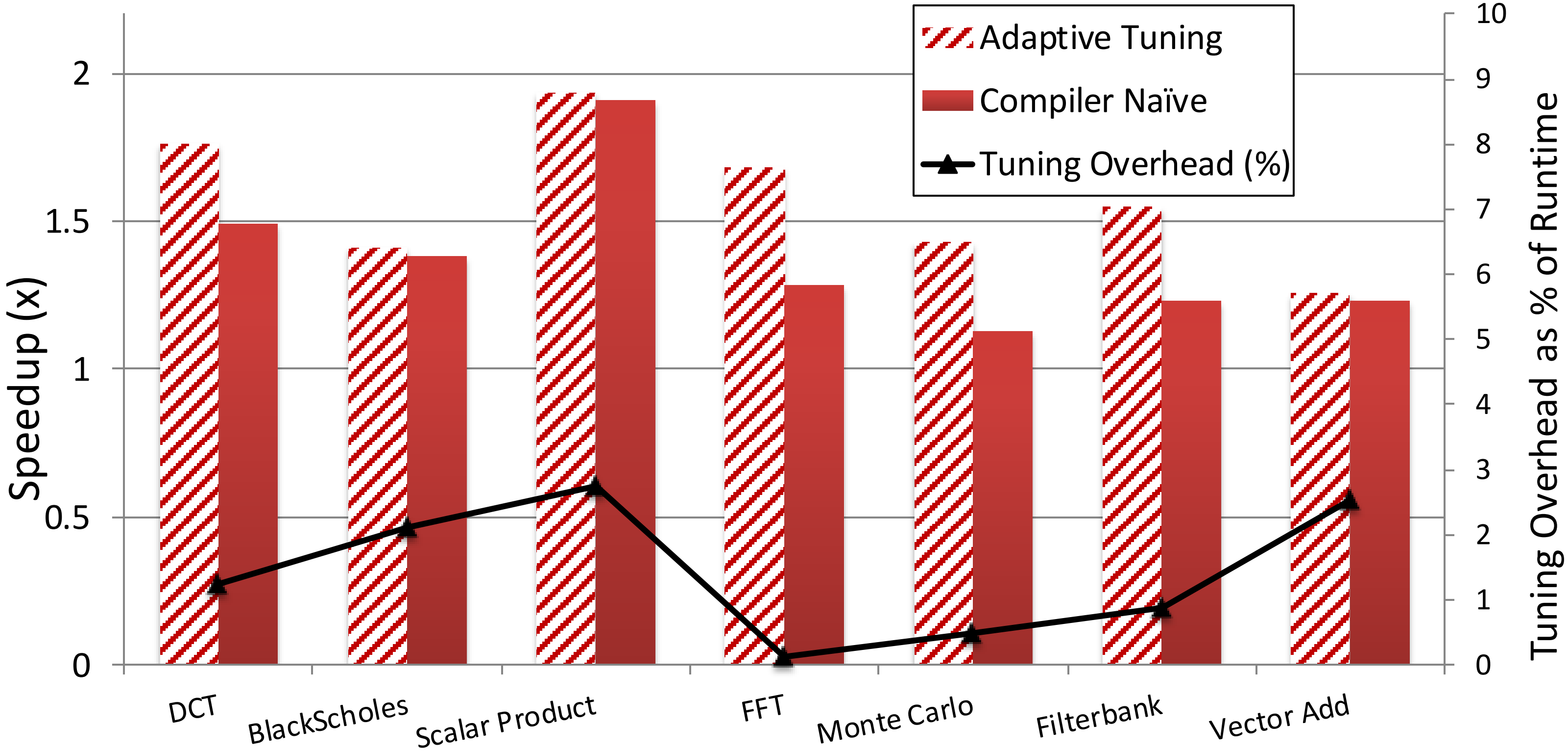
PURDUE
UNIVERSITY

Scalability

Benchmark	DataSize (Iteration Space)	CUDA Time (s)	OpenMPC Base Time (s)	%Copy-in Time(s)	%Copy-out Time(s)	%Kernel Time(s)	OpenMPC Pipelined Time (s)	Speedup Ideal	Speedup Achieved
Scalar Product	1024 x 1024	0.633	0.88807	52.19789	0.22676	47.57535	0.46297	1.91579	1.91822
	1024 x1024 x2	1.267	1.7723				0.91496		1.93703
	1024 x1024 x4	----	----				1.73067		
Monte Carlo	1024 x32	0.00537	0.00454	21.55109	13.71958	64.72933	0.0037	1.54489	1.22733
	1024 x1024 x32	***	1.79902				1.24699		1.44268
	1024 x1024 x64	***	3.5924				2.51465		1.42859
	1024 x1024 x128	----	----				5.02565		
Black Scholes	1024 x16	0.00105	0.00158	46.64777	41.9736	11.37863	0.00164	2.14373	0.96344
	1024 x1024 x128	1.8598	1.22591				0.87698		1.39786
	1024 x1024 x256	----	2.457				1.73985		1.41219
	1024 x1024 x384	----	----				2.60183		

`***' represent failure of the code due to larger-than-allowed grid sizes used. `----' represent code failure due to out-of-memory data size errors.

Tuning Performance

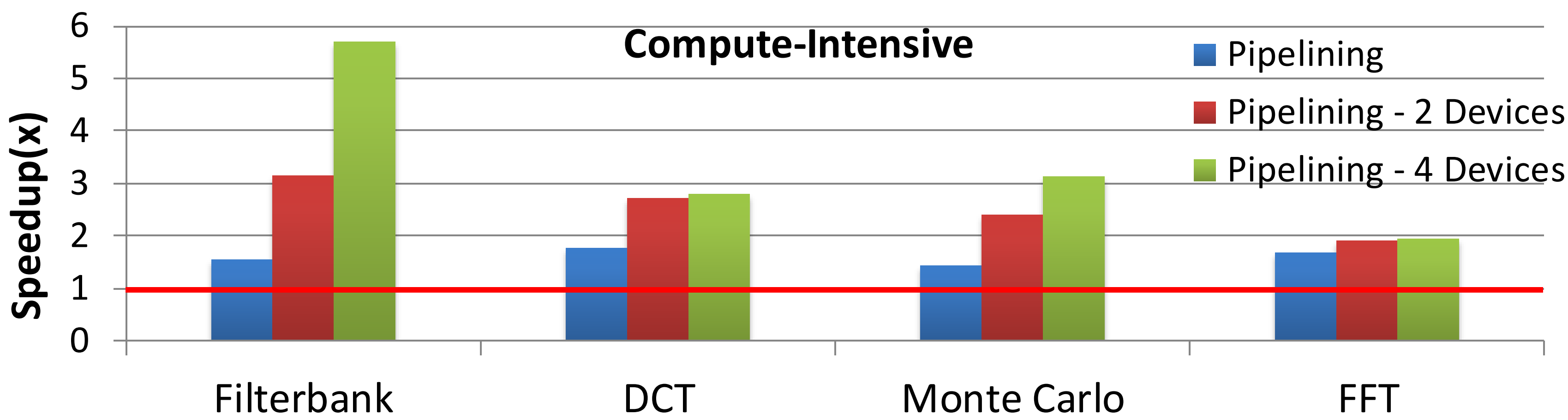


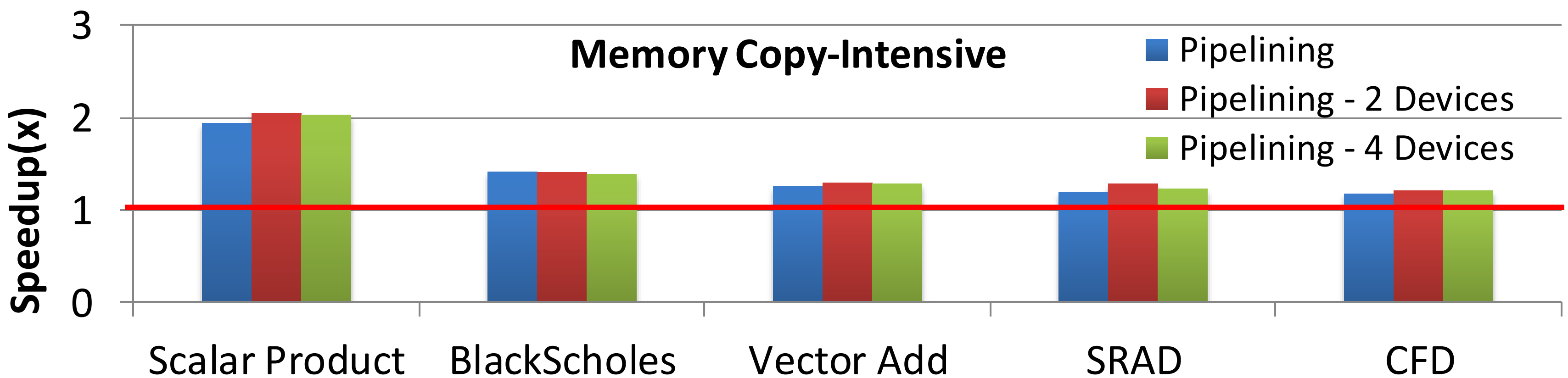
Naïve scheme – Use 1024 Splits (Heuristically found to be effective)

A static scheme to select SplitSize can not be efficient

PURDUE
UNIVERSITY

Comprehensive Results





- Compute-intensive benchmarks show better scalability on multi-GPU systems
- PCIe link forms a bottleneck on memory-copy intensive programs

PURDUE UNIVERSITY

Related Work

Out-of-card computations

- Single device image for multi-GPUs

Pipelining/Memory related

- Prefetch
- Redundant memory transfer removal
- Asynchronous computations

Execution Models

- StreamIt based approaches

Conclusions

We presented

- An automatic computation splitting mechanism, COSP, that handles out-of-card computations
- A mechanism to effectively pipeline the slow CPU-GPU data copy channels with GPU computation
- An automatic adaptive runtime tuning system to select optimal pipeline stage size
- A porting scheme to run OpenMP applications on multi-GPU systems

PURDUE
UNIVERSITY

Future Work

- Better strategy to deal with irregular applications
- Smart virtualization of the GPU address space – exploiting prediction to move data back and forth between CPU and GPU memories

PURDUE UNIVERSITY

Thank You!

PURDUE
UNIVERSITY