Efficient Deployment of Irregular Computations on Multi- and Many-core Architectures

Hancheng Wu, hwu16@ncsu.edu.com

NC STATE UNIVERSITY

Introduction

Multi- and Manycore processors have been advancing High Performance Computing with their high throughput and power efficiency. There has been an increasing interest in accelerating irregular computations on these devices that offer massive parallelism. This project proposes code transformations and compiler techniques that facilitate the deployment of irregular computations on multi- and many-core processors, aiming to achieve higher performance and to provide better programmability.

An Analytical Study of Recursive Tree Applications on Multi- and Manycore Processors

Identify and analyze parallel recursive tree traversal patterns, then select the best platform and code template based on traverse-specific characteristic.

Parallel Recursive Tree Traversal Patterns

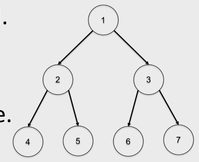
PTSD : a ***single*** parallelizable recursive tree traversal.

STMD: ***multiple*** serial tree traversals.

static - all traversals follow the same sequence.

dynamic - traversal order differs from node to node.

topdown - traversal visits one child per node.



STMD Patterns Optimization Variants

Code Variants	Method to Track Traversal Order	Characteristics
REC	Stack of each thread	Naïve parallelization
Dynamic Rope	Explicit user-defined stack	Iterative; Rope computed at runtime
Static Rope	Static pointers installed on the tree	Iterative; Rope computed only once avoid rec/user-stack

GPU Specific Optimizations:

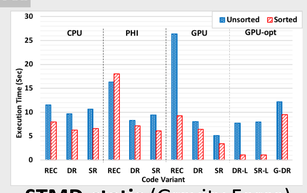
Greed DR - Allows a thread to fetch a new query immediately upon finish the current.

Lockstep - Force threads in a warp to follow the same traversal path.

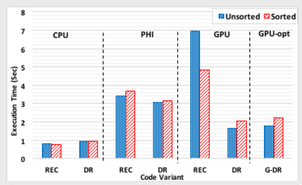
PTSD Pattern Optimization Variants

Code Variants	Advantage	Disadvantage
REC	Naïve Form	Threads Creation and Sync Overheads
Consolidated REC	Reduced # of REC calls	Buffering & Sync Overhead
FLAT	Flat Parallelism	Less work-efficient & Atomic

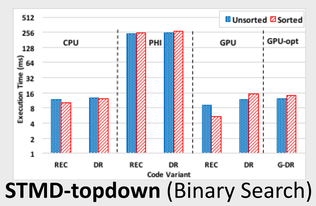
Results



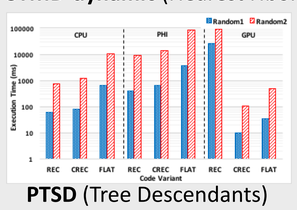
STMD-static (Gravity Force)



STMD-dynamic (Nearest Nbor)



STMD-topdown (Binary Search)



PTSD (Tree Descendants)

Summary

No single best solution that works for all

Pattern	Best Plat	Best Optimization Variant
STMD-static	GPU	Lockstep Execution + Rope + Sort
STMD-dynamic	CPU	Recursion or Rope
STMD-topdown	GPU	Recursion + Sort
PTSD	GPU	Consolidated Recursive Kernel

Compiler Based Workload Consolidation for Efficient Dynamic Parallelism on GPUs

Most irregular computations such as graph and tree algorithms can be expressed as patterns of irregular loops and parallel recursion. We propose a compiler framework to improve the efficiency of such irregular patterns written using Dynamic Parallelism by workload consolidation on GPUs.

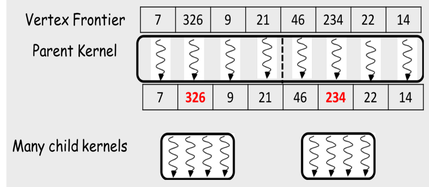
Basic Template of DP Kernel

```
__global__ void parent_kernel(){
  job = get_work_item();
  prework(job);
  if (condition)
    #pragma dp consldt(grid) buffer(default) work(job)
        child_kernel<<<block_dim, thread_dim>>>(…, job, …);
  else
    work (job);
  sync ();
  postwork(thread_id);
}
```

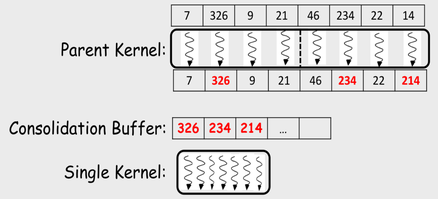
Kernel After Consolidation

```
__global__ void parent_kernel(){
  job = get_work_item(…);
  prework(job);
  if (condition) buffer.insert(job);
  else work(job);
  sync();
  if (thread_id==selected)
    child_kernel_consolidated<<<B, T>>>(buffer);
  sync();
  postwork(buffer)
}
```

Naïve DP Kernel Execution



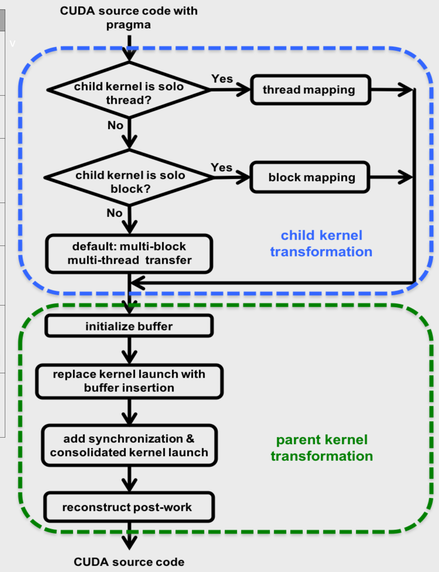
Grid-level Consolidation



Compiler Directives

Clause	Argument	Description
consldt	granularity: warp, block, grid	Workload consolidation granularity
buffer	type: default, halloc, custom	Buffer allocation mechanism
	perBufferSize: integer or variable name	Buffer size
	totalSize: integer	Total size of all buffers
work	varlist: list of indexes or pointers to work	List of variables to be stored in buffer
threads	thread number: integer	Number of thread/block for consolidated kernel
blocks	block number: integer	Number of blocks for consolidated kernel

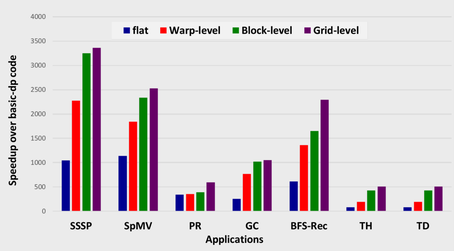
Transformation Steps



Postwork Transformation

- *Postwork* may depend on results from child kernels. Thus, synchronization b/w parent and child kernels become problematic for grid-level consolidation.
- Use customized global sync primitive.
- Correct kernel configuration to avoid deadlocks

Results



Grid-level Consolidation is the best with correct kernel configuration, yielding 90x to 3300x speedup over basic DP-based solutions and 2x to 6x speedup over flat implementations.

Acknowledgement

This work has been supported by NSF award CCF-1741683 and equipment donations from Nvidia Corporation.

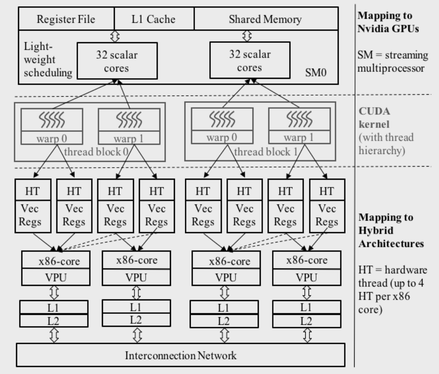
Compiling SIMT Programs on Multi- and Manycore Processors with Wide Vector Units

High performance on multi- and manycore (co)processors with wide vector extensions requires using both their x86-compatible cores and vector units (MIMD-SIMD **hybrid architectures**). We propose compiler techniques to deploy programs written using a SIMT programming model (a subset of CUDA C) on hybrid architectures. We point out the main challenges in supporting the SIMT model, especially for irregular computations on hybrid architectures.

The SIMD Model

The Subset of CUDA C Supported	
Types	**Operations**
Data Type	int, float (32-bit data)
Arithmetic Ops	+, -, *, /
Shifting Ops	<<, >>
Memory Allocation	cudaMalloc, cudaFree
Memory Transfer	cudaMemcpy
Assignments	a=b; a=b[tid]; b[gid]=a;
Synchronization	__syncthreads()
Vector Lane Identifier	gridDim, blockDim blockIdx, threadIdx

How To Map a CUDA Kernel



Microbenchmark and Results

```
1.  for_loop(num_iterations){
2.    for_each_thread{
3.      random_read(input_array,random_degree);
4.      random_write(output_array,random_degree);
5.  }
```

(a) Irregular memory accesses

```
6.  if(threadId<bd1){
7.    loop1(cost);
8.  }else if (threadId>=bd1 && threadId<bd2){
9.    loop2(cost);
10. }else if(...){
11.   ...
12. }else if (threadId>=bdD-1 && threadId<bdD){
13.   loopD(cost);
14. }
```

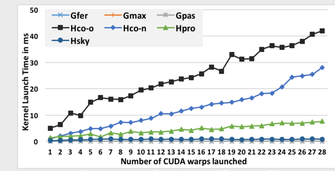
(b) Control divergence

```
15. for_loop(num_iterations){
16.   __syncthreads();
17. }
```

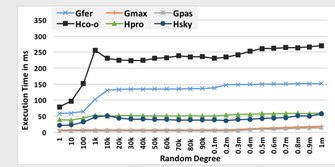
(c) Thread-block synchronization

```
18. for_loop(num_iterations){
19.   shared_mem_write(sm_array, random_degree);
20.   shared_mem_read(sm_array, random_degree);
21. }
```

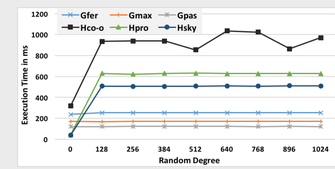
(d) Shared memory accesses



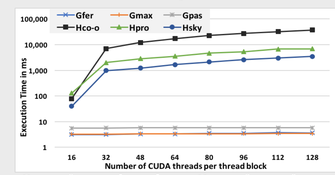
Kernel Launch Overhead



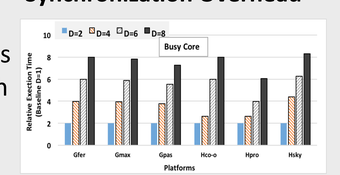
Memory R&W access time



Shared Memory Performance

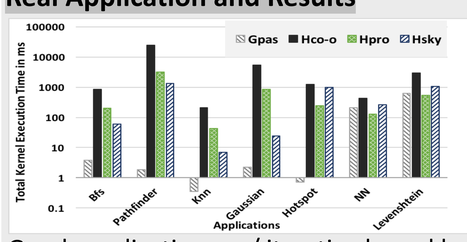


Synchronization Overhead



Control Divergence Overhead

Real Application and Results



Graph applications w/ iterative kernel launches can hardly benefit. Irregular computations such as BFS will suffer. Computations with regular memory accesses, few kernel invocations and no synchronization can run faster on hybrid architectures than on GPUs (NN).