

Rodinia: A Benchmark Suite for Heterogeneous Computing

Kevin Skadron and Shuai Che
University of Virginia

Agenda

- Benchmarking for heterogeneous platforms
- Brief overview of heterogeneous architecture issues, CUDA and its optimization strategies
- Overview of the *Rodinia* benchmark suite
- Discussion

Benchmarking

- Role of benchmarking
 - Help designers to explore architectural design
 - Identify bottlenecks
 - Compare different systems
 - Use benchmarks to conduct performance prediction
- Requirements
 - Demonstrate diverse behaviors in terms of both program characteristics and how they stress arch components
 - Span sufficiently large application space
 - Depend on research needs (general purpose vs. embedded architectures)

Related Work

- Benchmark suites
 - SPEC, SPLASH-2, Parsec, EEMBC ...
 - ALPBench, MediaBench, BioParallel ...
 - Parboil for GPU
- Benchmark analysis and design
 - Workload characterization
 - Redundancy
 - Performance prediction




A Suite for Heterogeneous Computing

- Accelerators, such as GPUs and FPGAs, are emerging as popular platforms for computing
 - High performance
 - Energy efficiency
 - Improving programmability
- Research issues
 - What problems can accelerators solve with high performance and efficiency?
 - How to optimize both CPU and accelerators to best work together on various workloads?
 - What features of programming model and hardware architecture are needed?
- We need a diverse set of applications
 - The *Rodinia* suite supports multicore CPU and GPU

















Motivations in Designing *Rodinia*

- Program behaviors that are well/poorly suited to GPUs have not been systematically explored
- Architects need diverse applications to help decide what hardware features should be included in the limited area budgets
- Diverse implementations for GPUs provide exemplars for different types of applications, assisting in the porting of new applications

Parsec vs. SPLASH-2. vs. Rodinia

	Parsec	SPLASH-2	Rodinia
Platform	CPU	CPU	CPU and GPU
Programming Model	Pthreads, OpenMP, TBB	PARMACS macros	OpenMP, CUDA
Machine Model	shared memory	shared memory	shared memory, offloading
Application Domain	scientific/engineering, finance, multimedia	scientific/engineering, graphics	scientific/engineering, data mining
NO. of Applications	3 kernels, 9 apps	4 kernels and 8 apps	6 kernels and 5 apps
Optimized for	multicore	distributed shared memory multiprocessor	manycore, accelerator
Incremental Opt. Ver.			
Memory Space	HW cache	HW cache	HW/SW cache
Problem Sizes	small - large	small - medium	small - large
Special SW techniques	SW pipelining	NA	ghost-zone, persistent thread-block
Synchronization	barrier/lock/condition	barrier/lock/condition	barrier

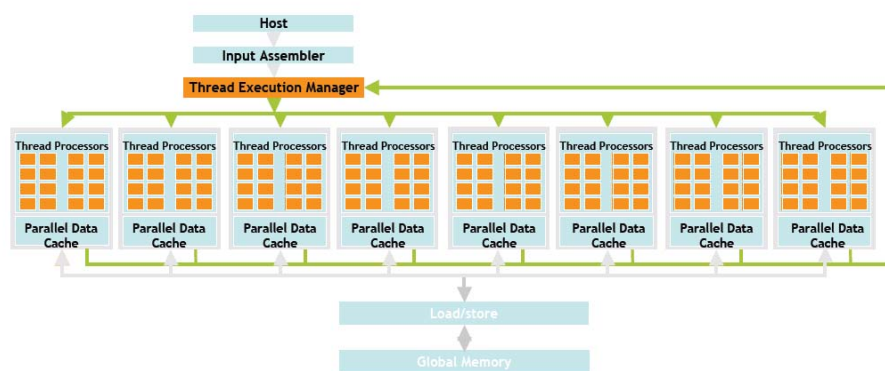
Other Benchmark Suites

	Multithreaded	Domain-specific	Model	Platform
SPEC CPU 2006			NA	CPU
SPEC OMP 2001			OpenMP	CPU
ALPBench			Pthreads	CPU
Biobench			NA	CPU
BioParallel			OpenMP	CPU
MediaBench			NA	CPU
MineBench			OpenMP	CPU
Parboil			CUDA	GPU

Discussion

- Benchmarking needs? Are existing benchmarks well designed?
- How to define a *kernel* versus an *application*?
- Is porting the existing suites (e.g. Parsec, Splash2) to the new platforms enough?
- What kind of features do you need for your research?
- What metrics do you care about?

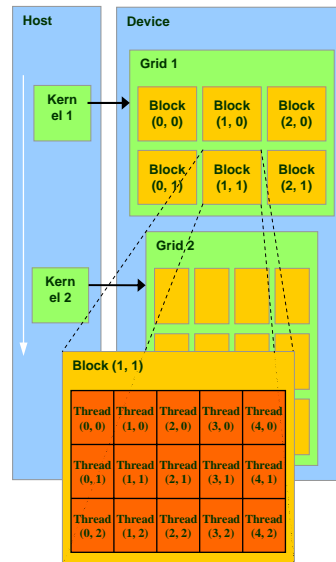
NVIDIA GPU Architecture



- Streaming Multiprocessor (SM), Streaming Processor (SP)
- Specialized memory spaces

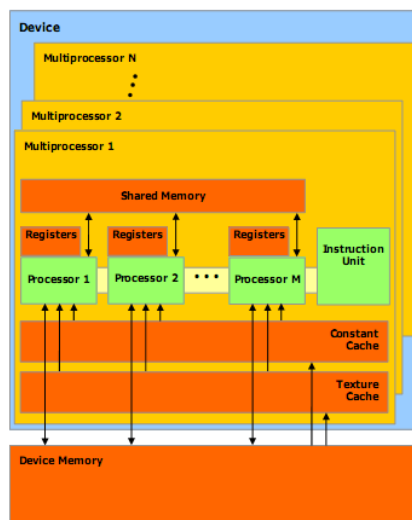
CUDA's Domain Based Model

- Hierarchical Model
 - CPU launch kernels with large number of threads
 - Single Instruction Multiple Threads (SIMT)
 - Computation Domain
 - Grid -> Block -> Warp -> Threads
 - Synchronization within a thread block



Images are cited from NVIDIA CUDA Programming Guide

Memory Hierarchy



- Relaxed memory consistency model
- Each thread can:
 - R/W per-block **shared memory**
 - R/W per-grid **global memory**
 - RO per-grid **constant memory**
 - RO per-grid **texture memory**

Images are cited from NVIDIA CUDA Programming Guide

Vector Add: CUDA Host Code

```
float *GPU_add_vectors(float *A_CPU, float *B_CPU, int N) {  
    // Allocate GPU memory for the inputs and the result  
    int vector_size = N * sizeof(float);  
    float *A_GPU, *B_GPU, *C_GPU;  
    cudaMalloc((void **) &A_GPU, vector_size);  
    cudaMalloc((void **) &B_GPU, vector_size);  
    cudaMalloc((void **) &C_GPU, vector_size);  
    // Transfer the input vectors to GPU memory  
    cudaMemcpy(A_GPU, A_CPU, vector_size, cudaMemcpyHostToDevice);  
    cudaMemcpy(B_GPU, B_CPU, vector_size, cudaMemcpyHostToDevice);  
    // Execute the kernel to compute the vector sum on the GPU  
    dim3 grid_size = ...  
    add_vectors_kernel <<< grid_size, threads_per_block >>> (A_GPU, B_GPU, C_GPU, N);  
    // Transfer the result vector from the GPU to the CPU  
    float *C_CPU = (float *) malloc(vector_size);  
    cudaMemcpy(C_CPU, C_GPU, vector_size, cudaMemcpyDeviceToHost);  
    return C_CPU;  
}
```

Vector Add: CUDA Kernel Code

```
// GPU kernel that computes the vector sum C = A + B  
// (each thread computes a single value of the result)  
__global__ void add_vectors_kernel(float *A, float *B, float *C, int N) {  
  
    // Determine which element this thread is computing  
    int block_id = blockIdx.x + gridDim.x * blockIdx.y;  
    int thread_id = blockDim.x * block_id + threadIdx.x;  
  
    // Compute a single element of the result vector (if it is valid)  
    if (thread_id < N) C[thread_id] = A[thread_id] + B[thread_id];  
}
```

Vector Add: OpenMP Code

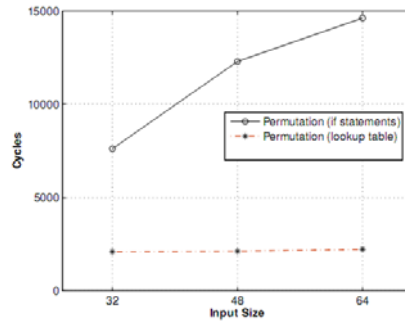
```
float *CPU_add_vectors(float *A, float *B, int N) {  
  
    // Allocate memory for the result  
    float *C = (float *) malloc(N * sizeof(float));  
  
    // Compute the sum;  
    #pragma omp parallel for  
    for (int i = 0; i < N; i++) C[i] = A[i] + B[i];  
  
    // Return the result  
    return C;  
}
```

Basic CUDA Optimizations

- Minimize data transfer between the host and the GPU
 - Asynchronous transfers and overlapping transfers with computation
 - Pinned memory
 - Algorithm changes to avoid memory transfer

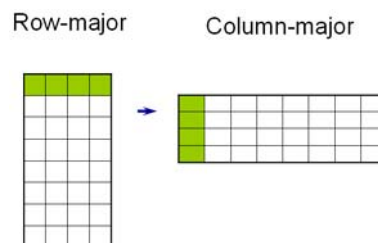
Basic CUDA Optimizations

- Minimize the usage of control-flow operations
 - The DES example (*if* statements vs. lookup table)



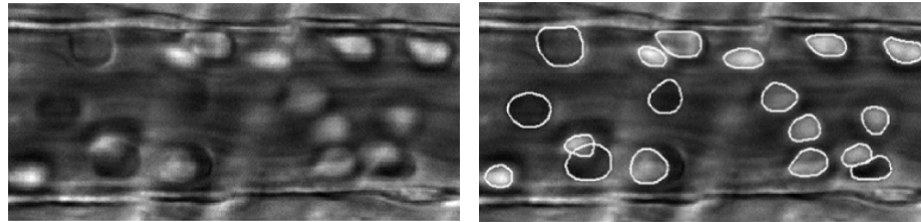
Basic CUDA Optimizations

- Take advantage of different GPU memory spaces
 - Shared memory (bank conflict)
 - Texture and constant memory
- Coalesced memory accesses
- An example:



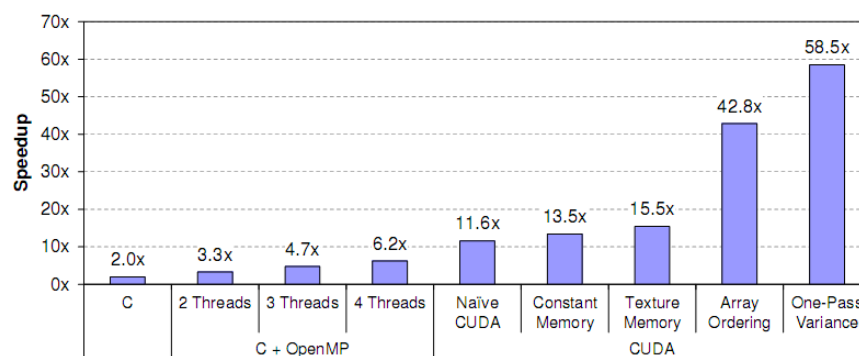
Leukocyte

- Boyer et. al. (IPDPS'09)
- Leukocyte detects and tracks rolling leukocytes (white blood cells) in vivo video microscopy of blood vessels



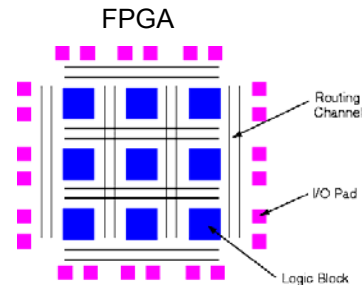
Optimizations

- Demo (CPU vs. GPU detection)



A GPU vs. FPGA Comparison

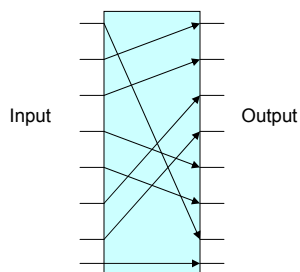
- GPU:
Fixed design with high throughput and memory bandwidth
- FPGA:
Uncommitted logic arrays that can approximate customized design
- Applications:
 - Gaussian Elimination
 - Needleman Wunsch
 - Data Encryption Standard (DES)



S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," IEEE Design and Test of Computers, 1996.

Implementation Example

- Our implementations of the three applications on FPGAs and GPUs are algorithmically similar
- For example: DES bit-wise permutation



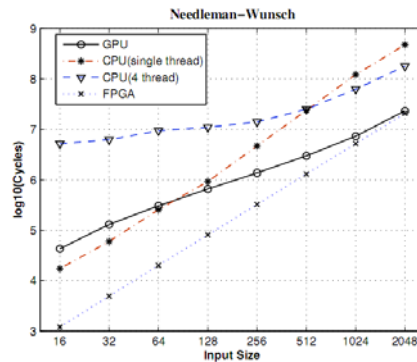
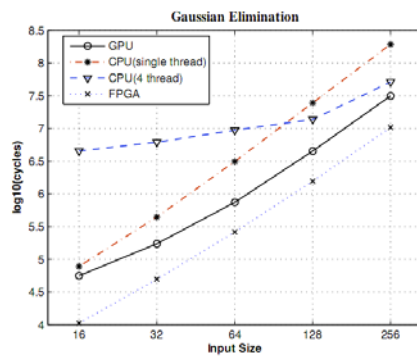
```
_global_ void permutation_kernel(int *lookup_table_device ...){  
    //lookup table is saved in the device memory  
    int tx= threadIdx.x; //thread id  
    __shared__ char per_source[DATA_LENGTH];  
    __shared__ char per_destination[DATA_LENGTH];  
    ...  
    //each thread is responsible for copying one data  
    permu_destination[tx] = permu_source[lookup_table_device[tx]];  
    ...  
}
```

FPGA hardware module

CUDA Code

Performance

- GPU implementations show their relative advantage for large data sets
- The FPGA usually has the lowest overhead, but if taking *clock frequency* into account, it is not necessarily the fastest.



Mapping Applications to Accelerators

Fit Good

GPU

FPGA

• No interdependency and massively parallel (*Gaussian Elimination*).

• Functions which are not directly supported by general purpose instruction sets (*DES*).

• Operations that can not be efficiently implemented on the GPU (*DES*).

• Implementations can take advantage of streaming and pipelining (*Needleman-Wunsch*).

• Limited parallelism and low arithmetic intensity (*Needleman-Wunsch*)

• Require a lot of complexity in design (*Gaussian Elimination*).

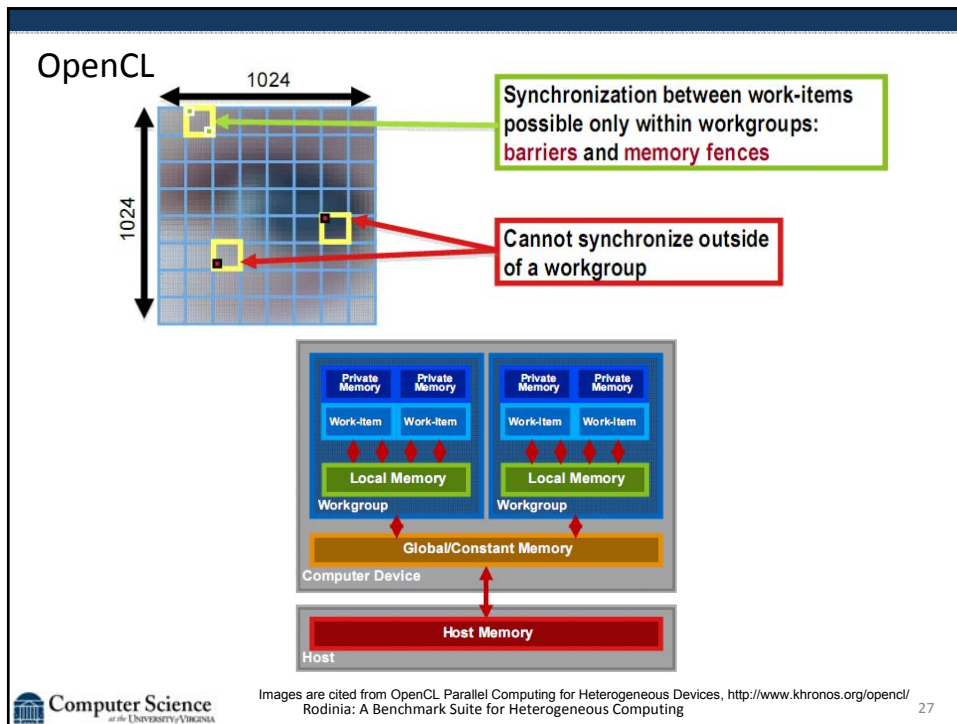
Fit Bad

Challenges of Heterogeneous Systems

- Metrics for fair comparison
 - Other physical behaviors are equally important such as power, temperature, QoS, etc.
- Allocate tasks to the best-fit cores
- How many core types do we need?
- Models for different architectures
 - Things that make the world complicated:
Different types of cores, memory hierarchy, coherence protocol, die size, etc.
- The balance between SW and HW design

OpenCL

- A unified framework for various platforms
- Similar to CUDA
- The techniques we used for *Rodinia* will be easily translated into OpenCL
- Work items, work groups, global memory space, synchronization in per-work-group storage



OpenCL vs. CUDA

- Revisit vectorAdd()

C for CUDA Kernel Code:

```
__global__ void
vectorAdd(const float * a, const float * b, float * c)
{
    // Vector element index
    int nIndex = blockIdx.x * blockDim.x + threadIdx.x;

    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL Kernel Code

```
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);

    c[nIndex] = a[nIndex] + b[nIndex];
}
```

The code is from NVIDIA Corporation, NVIDIA OpenCL JumpStart Guide, April 2009

Rodinia: A Benchmark Suite for Heterogeneous Computing

OpenCL Host Code

```
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks    = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                   0, 0, 0);

// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
                 0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
                 nContextDescriptorSize, aDevices, 0);

// create a command queue for first device the context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
                                     sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);
```

OpenCL Host Code

```
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd", 0);

// allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];

// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);

// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float),
                             pA,
                             0);
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             cnDimension * sizeof(cl_float),
                             pA,
                             0);
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             cnDimension * sizeof(cl_float),
                             0, 0);
```

OpenCL Host Code

```
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);

// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                      &cnDimension, 0, 0, 0, 0);

// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                  cnDimension * sizeof(cl_float),
                  pC, 0, 0, 0);

delete[] pA;
delete[] pB;
delete[] pC;

clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

Questions and Discussions

- What is the best way to deal with the portability and legacy code issues? New languages (e.g. OpenCL) or old languages (e.g. OpenMP, PGI) with compiler support?
- What is the best way to deal with large code bases?
- If software is optimized for specific hardware details, how to deal with rapid evolution?

The *Rodinia* Benchmark Suite

- Five applications and seven kernels
 - CUDA for GPUs and OpenMP for multicore CPUs
 - Various optimization techniques
- *Berkeley Dwarfs* are used as guidelines to choose applications with diverse characteristics.
 - Discussion: are the dwarfs a good taxonomy for application classification?
- A diverse range of application domains
- Different versions for several applications by applying incremental optimizations

Rodinia Applications

Applications	Dwarfs	Domains
Leukocyte Det./Track	Structured Grid	Medical Imaging
SRAD	Structured Grid	Physics Simulation
HotSpot	Structured Grid	Image Processing
Back Propagation	Unstructured Grid	Pattern Recognition
Needleman Wunsch	Dynamic Programming	Bioinformatics
K-means	Dense Linear Algebra	Data Mining
Streamcluster	Dense Linear Algebra	Data Mining
Breadth-First Search	Graph Traversal	Graph Algorithms
Heartwall tracking	Structured Grid	Medical Imaging
MUMmerGPU	Graph Traversal	Bioinformatics
LU Decomposition	Dense Linear Algebra	Linear Algebra
CFD	Unstructured Grid	Fluid Dynamics

Use *Rodinia*

- Download site
 - <http://lava.cs.virginia.edu/wiki/rodinia>
- Can be used for running on both native GPUs and simulators such as GPGPUsim (ISPASS'09)
- We are preparing an OpenMP package compatible with the M5 simulator
- *Rodinia* is useful in
 - Manycore architecture research
 - Providing versions with successive layers of optimization, allowing designers to evaluate the impact of ways of parallelization on architectural design

A *Rodinia* Sub-Suite Hard for Compilers

- We provide a list of applications that may be relatively hard for compilers to automatically generate GPU codes
- Applications: *Needleman Wunsch*, *LU Decomposition*, *Myocyte*, *Heartwall* and *Leukocyte*
- Some applications have multiple levels of parallelism that is hard to detect
- Useful for comparing compiler generated code with manually optimized *Rodinia* code

Rodinia Usage

- Package Structure

```
rodinia_1.0/bin      : binary executables
rodinia_1.0/common   : common configuration file
rodinia_1.0/cuda     : source code for the CUDA implementations
rodinia_1.0/data     : input files
rodinia_1.0/openmp   : source code for the OpenMP implementations
```

- Modify common/make.config

```
#Rodinia home directory
RODINIA_HOME = ~sc5nz/rodinia_1.0

#CUDA binary executable
CUDA_BIN_DIR=${RODINIA_HOME}/bin/linux/cuda

#OpenMP binary executable
OMP_BIN_DIR=${RODINIA_HOME}/bin/linux/omp

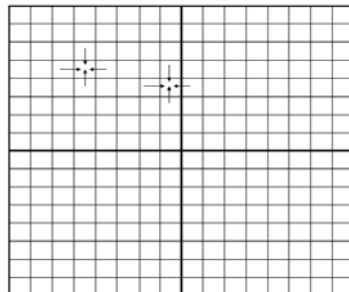
# CUDA toolkit installation path
CUDA_DIR = /usr/local/cuda

# CUDA SDK installation path
SDK_DIR = /af10/sc5nz/NVIDIA_CUDA_SDK
```

- In the *Rodinia* home directory, simply type *make* to build

Case I: HotSpot

- A widely used tool to estimate processor temperature
- Dwarf: *Structured Grid*
- Each cell in the 2-D grid represents the average temperature value of the corresponding area of the chip

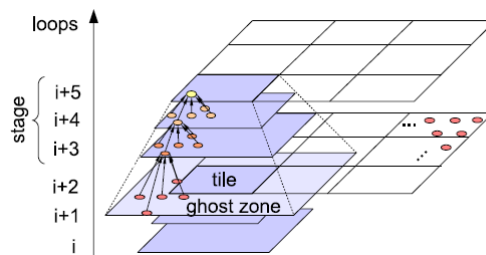


Naïve Implementation

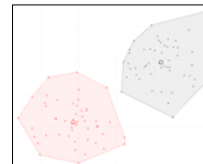
- Each data block is mapped to one thread block
- The boundary data is exchanged between two iterations
- Poor performance
 - global memory accesses
 - synchronization via kernel call

Ghost-Zone

- Use a ghost zone of redundant data around each block to minimize global communication overhead
- If the base is an $N \times N$ data block, then after one iteration, the inner $(N - 2) \times (N - 2)$ data block contains valid results.
- Take advantage of the low-latency shared memory and reduce kernel-call overhead



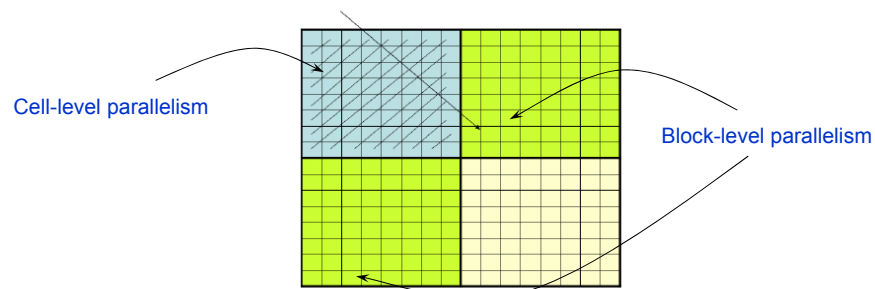
Case II: Kmeans



- A widely used clustering algorithm
- Dwarf: *Dense Linear Algebra*
- Procedures
 - Initial k cluster centers are chosen
 - Associate each data object to closest centers
 - Recalculate centers by taking the mean of all data objects in clusters
 - Repeat until no objects move from one cluster to another
- CUDA implementation
 - Data objects are partitioned into thread blocks, with each thread associated with one data object
- Optimizations
 - The array holding the centers is organized to fit in the *constant memory*
 - The read-only main array is bound to a texture to take advantage of the *texture memory*

Case III: Needleman-Wunsch

- A global optimization method for DNA sequence alignment
- Dwarf: *Dynamic Programming*
- Two major phases:
 - 1) It fills the matrix with scores in a diagonal-strip manner (parallel)
 - 2) A trace-back process finds the optimal alignment (sequential)
- Optimizations: two levels of parallelism
 - Data elements on the same diagonal within a thread block
 - Thread blocks on the same diagonal within the overall matrix



Code Walk-Through (NW Host)



```

cudaMalloc((void**)& reference_cuda, sizeof(int)*size);
cudaMalloc((void**)& matrix_cuda, sizeof(int)*size);

cudaMemcpy(reference_cuda, reference, sizeof(int) * size, cudaMemcpyHostToDevice);
cudaMemcpy(matrix_cuda, input_itemsets, sizeof(int) * size, cudaMemcpyHostToDevice);

dim3 dimGrid;
dim3 dimBlock(BLOCK_SIZE, 1);
int block_width = ( max_cols - 1 )/BLOCK_SIZE;

printf("Processing top-left matrix\n");

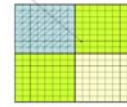
for( int i = 1 ; i <= block_width ; i++){
    dimGrid.x = i;
    dimGrid.y = 1;
    needle_cuda_shared_1<<<dimGrid, dimBlock>>>(reference_cuda, matrix_cuda,
                                                ,max_cols, penalty, 1, block_width);
}

printf("Processing bottom-right matrix\n");

for( int i = block_width - 1 ; i >= 1 ; i--){
    dimGrid.x = i;
    dimGrid.y = 1;
    needle_cuda_shared_2<<<dimGrid, dimBlock>>>(reference_cuda, matrix_cuda,
                                                ,max_cols, penalty, 1, block_width);
}

cudaMemcpy(output_itemsets, matrix_cuda, sizeof(int) * size, cudaMemcpyDeviceToHost);
  
```

Code Walk-Through (NW kernel)



```
__shared__ int temp[BLOCK_SIZE+1][BLOCK_SIZE+1];
__shared__ int ref[BLOCK_SIZE][BLOCK_SIZE];

if (tx == 0)
    temp[tx][0] = matrix_cuda[index_nw];

for ( int ty = 0 ; ty < BLOCK_SIZE ; ty++)
    ref[ty][tx] = reference[index + cols * ty];
__syncthreads();

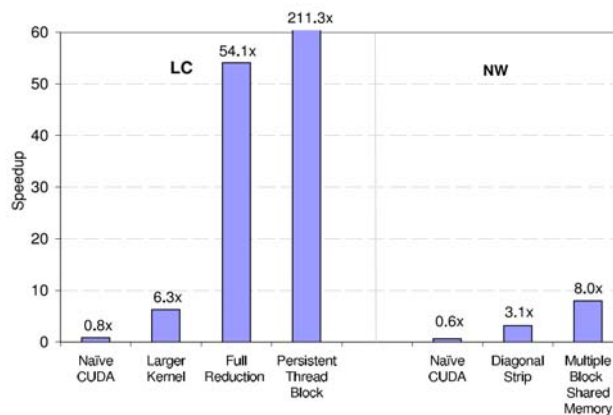
temp[tx + 1][0] = matrix_cuda[index_w + cols * tx];
__syncthreads();
temp[0][tx + 1] = matrix_cuda[index_n];
__syncthreads();

for( int m = 0 ; m < BLOCK_SIZE ; m++){
    if ( tx <= m ){
        int t_index_x = tx + 1;
        int t_index_y = m - tx + 1;
        temp[t_index_y][t_index_x] = maximum( temp[t_index_y-1][t_index_x-1] + ref[t_index_y-1][t_index_x-1],
                                                temp[t_index_y][t_index_x-1] - penalty, temp[t_index_y-1][t_index_x] - penalty);
    }
    __syncthreads();
}

for( int m = BLOCK_SIZE - 2 ; m >= 0 ; m--){
    if ( tx <= m ){
        int t_index_x = tx + BLOCK_SIZE - m ;
        int t_index_y = BLOCK_SIZE - tx;
        temp[t_index_y][t_index_x] = maximum( temp[t_index_y-1][t_index_x-1] + ref[t_index_y-1][t_index_x-1],
                                                temp[t_index_y][t_index_x-1] - penalty, temp[t_index_y-1][t_index_x] - penalty);
    }
    __syncthreads();
}

for ( int ty = 0 ; ty < BLOCK_SIZE ; ty++)
    matrix_cuda[index + ty * cols] = temp[ty+1][tx+1];
```

Incremental Performance Improvement

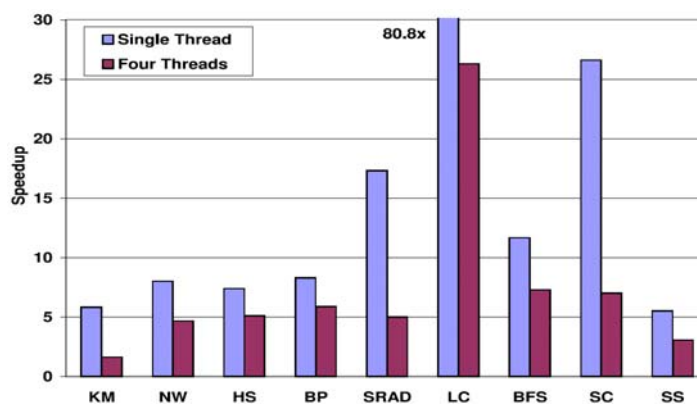


- Which optimization to apply and the order to apply optimizations is not always intuitive

Experience Using CUDA and Lesson Learned

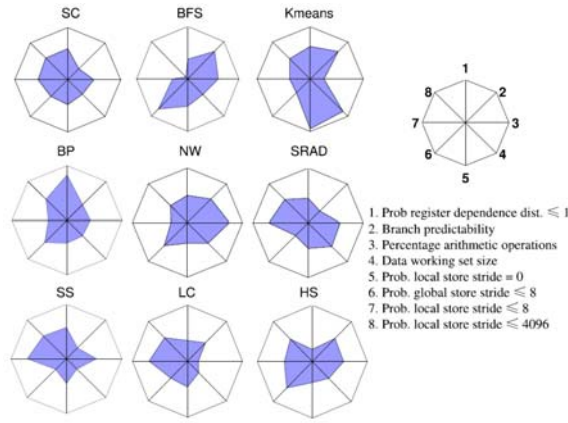
- Mappings of applications' data structures to CUDA's domain-based model
- Non-intuitive algorithmic optimization techniques
- Memory access patterns
- Global memory fence
- Offloading decision considering various overheads
- Tradeoff between single-thread performance and parallel throughput

Performance



- GPU: NVIDIA Geforce GTX 280 (1.3 GHz shader clock)
- CPU: Intel Quad-core Intel Core 2 Extreme (3.2 GHz)
- Speedups: 5.5 ~ 80.8 (single-thread) and 1.6 ~ 26.3 (four-thread)

Diversity Analysis



- **Microarchitecture-Independent Workload Characterization (MICA)** by Hoste and Eeckhout (*IISWC'06*)
- **Metrics:** Instruction mix, Register traffic, Working set, Data-stream size and Branch-predictability, etc.

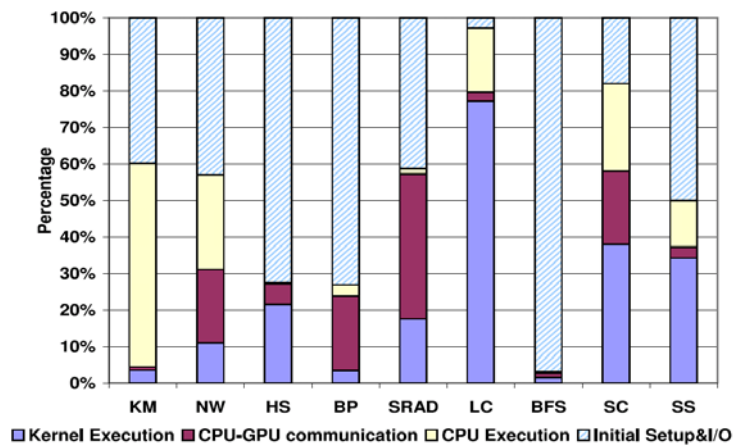
Rodinia Statistics

	KM	NW	HS	BP	SRAD	LC	BFS	SC	SS
Kernels	2	2	1	2	2	3	2	1	14
Barriers	6	70	3	5	9	7	0	1	15
Lines of Code	1100	430	340	960	310	4300	290	1300	100
Optimizations	C/CA/S/T	S	S/Pyramid	S	S	C/CA/T		S	S/CA
Problem Size	819200 points 34 features	2048 × 2048 data points	500 × 500 data points	65536 input nodes	2048 × 2048 pixels/frame	219 × 640 pixels/frame	10 ⁶ nodes	65536 points 256 dimensions	256 points 128 features
CPU execution time	20.9 s	395.1 ms	3.6 ms	84.2 ms	40.4 s	122.4 s	3.7 s	171.0 s	33.9 ms
L2 Miss Rate (%)	27.4	41.2	7.0	7.8	1.8	0.06	21.0	8.4	11.7

C = Constant Memory; CA = Coalesced Memory Access; T = Texture; S = Shared Memory

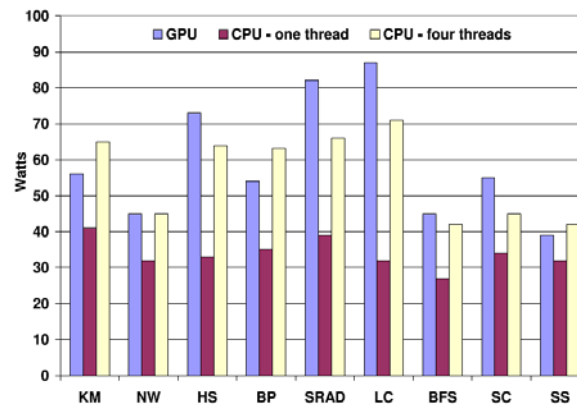
- Resource usage
- Problem size
- Number of kernels and threads
- Optimization techniques
 - Hardware-level optimizations
 - *Shared*, *Constant* and *Texture* memory, and *Coalesced Access*
 - Algorithm-level optimizations
 - Ghost Zone (*HotSpot*), Persistent Thread Block (*Leukocyte*)

Breakdown of Execution Time



- Memory copying: 2%-76%, excluding I/O and initial setup
 - Due to CPU phases between GPU kernels: *SRAD, Back Propagation*
 - All the computations are done on the GPU: *Needleman Wunsch, HotSpot*

Power Consumption



- GPU power consumption: 38 ~ 87 W (186 W idle power)
- The power-performance efficiency almost always favors the GPU

Comparison with Parsec

- Major features evaluated:
instruction mix, working set, and sharing behavior
- A *Pintool* is used to collect program characteristics
- Clustering analysis is based on distances between applications
- Principal component analysis (PCA) is used to identify important characteristics

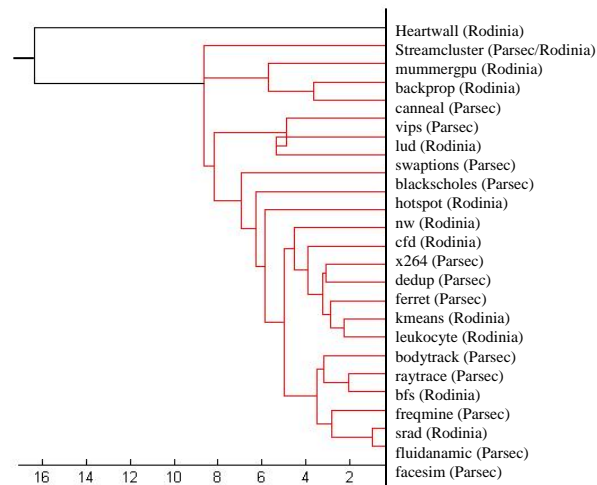
$$Z_i = \sum_{j=1}^p a_{ij} X_j$$
$$X_1, X_2, \dots, X_p \longrightarrow Z_1, Z_2, \dots, Z_p$$

$$(i) \text{Var}[Z_1] \geq \text{Var}[Z_2] \geq \dots \geq \text{Var}[Z_p]$$

$$(ii) \text{Cov}[Z_i, Z_j] = 0, \forall i \neq j$$

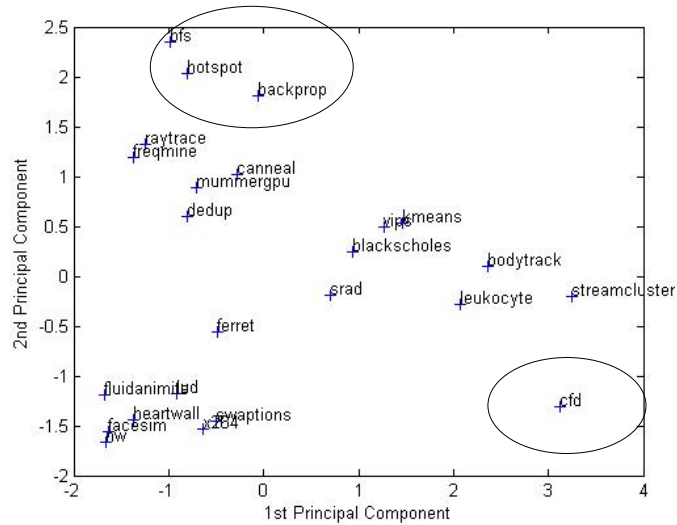
Comparison with Parsec

- Parsec and *Rodinia* span almost similar application space



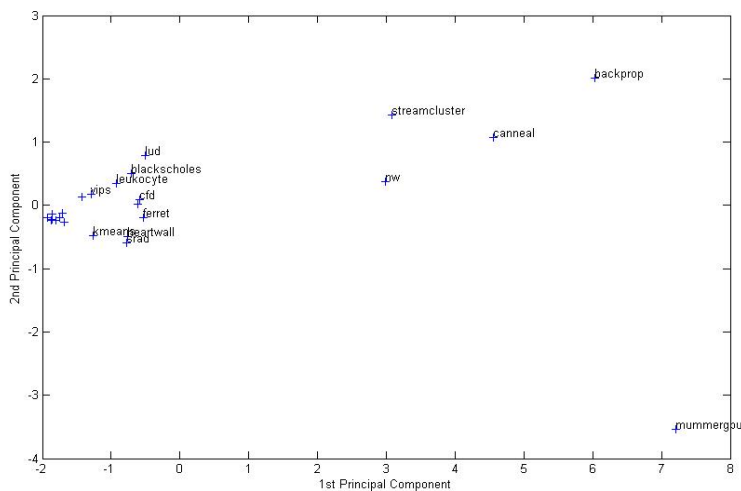
Instruction Mix

- Rodinia* demonstrates instruction mix features not in Parsec



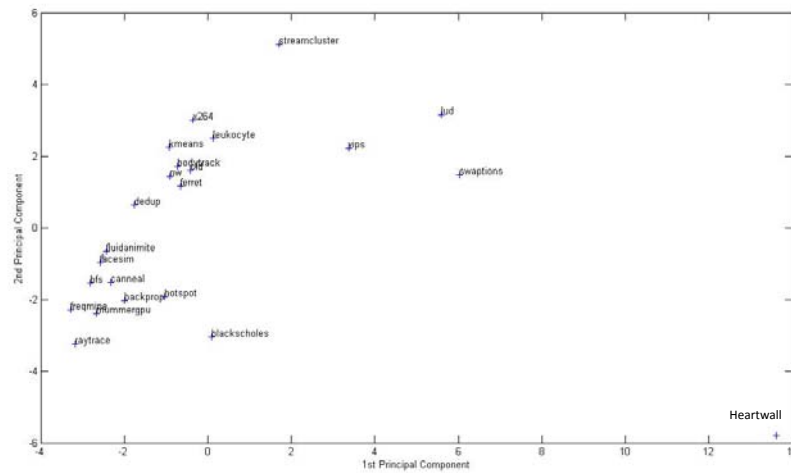
Working Set

- Mummer* in *Rodinia* is significantly different from the rest



Sharing

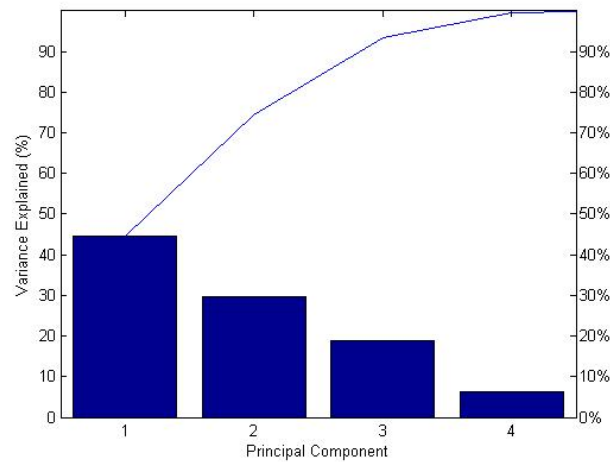
- Heartwall in *Rodinia* is significantly different from the rest



Confidence

$$\sum_{i=1}^p \text{Var}[X_i] = \sum_{i=1}^p \text{Var}[Z_i]$$

$$\text{Variance Explained} : (\sum_{i=1}^q \text{Var}[Z_i]) / (\sum_{i=1}^p \text{Var}[X_i])$$



General Discussions about Benchmarking

- How to make benchmark suites more useful? What are potential research questions in benchmarking?
- How optimized? Average or Code Hero?
- Stressmarks, building blocks, standalone applications, workflows?
- What languages?
- What is the new “general purpose”? (supercomputing, cloud computing, mobile platforms, netbooks, etc.)
- What are the new metrics?
- How to make it more useful for prediction?
- Any comments on the *Rodinia* design, its To-Do list and potential usage?
- A small exercise:

What is your top-3 wish list for what is missing in current multicore/heterogeneous benchmarking?

Conclusion

- The *Rodinia* benchmarks exhibit diverse application characteristics
- Rodinia is the first suite that allows comparison between multicore CPU and GPU
- Future work includes:
 - Add new applications, including the ones with poor GPU performance
 - Include more inputs representing diverse behavior
 - Extend Rodinia to support more platforms
 - Develop architecture-independent metrics and tools to compare different platforms

Thank you!

Please visit

<http://lava.cs.virginia.edu/wiki/rodinia>