Rodinia Benchmark Suite

CIS 601 Paper Presentation
3/16/2017

Presented by Grayson Honan, Shreyas Shivakumar, Akshay Sriraman

Rodinia: A Benchmark Suite for Heterogeneous Computing
Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee and Kevin Skadron

Introduction

- What is Rodinia?
  - Benchmarking suite for heterogeneous computing
  - Applications (inspired by Berkeley's dwarf taxonomy) and kernels to run on multi-core CPUs (OpenMP), GPUs (CUDA) *and* ****OpenCL***
- Why?
  - Standard benchmark program to compare platforms
  - Identify performance bottlenecks, evaluate solutions and study emerging platforms (GPUs)
  - Illustrate architectural differences between CPUs and GPUs

Introduction

- How?
  - By quantitatively measuring parallel communication patterns, synchronization techniques, power consumption and the effect of data layouts and bandwidth limitations
  - Each application / kernel is chosen to represent different types of behaviour - ***Berkeley Dwarves*** *(*9 dwarves at the time of writing)*
- *Quick Disclaimer : This paper was written in 2009, and much has changed since. We have tried to include as much up to date information as possible.*

Q1. What are Berkeley Dwarves?

Introduction

- **Berkeley Dwarves:**
  - Algorithmic method that captures a **pattern of computation and/or communication**
  - Specified at **high levels of abstraction** to allow reasoning across a **broad range** of applications
  - Implementations may be different but the **underlying patterns will persist** through generations of changes
  - **While they are useful guiding principles, may not sufficiently ensure adequate diversity.*

Introduction

List of Dwarfs

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

K-Means

Particle Filters

Back Propagation

Monte Carlo Simulations

Breadth First Search

Knapsack Problem

Huffman Encoding

Introduction

- Observations (GPU)
  - Low ratio of on-chip storage to #threads
  - Compensated for with **specialized memory spaces** (Shared Memory, Constant Memory, Texture Memory)
  - **Lack of persistence in Shared Memory** is less efficient for communication between kernels
  - No easy way for run-time load balancing among threads
  - High kernel-call and **data-transfer costs**

Q2. I see that some of these benchmarks use texture memory too. I thought texture memory was only used for graphics applications. What is texture memory and how does it differ from constant memory?

Texture Memory

- Read only - cached memory
- Traditionally designed for graphics
- Memory is stored on chip, provides higher effective bandwidth
- Used when you read memory often
- Large datasets, spatial locality read access patterns
    - *“ The first thing to keep in mind is that texture memory **is global memory**. The only difference is that textures are accessed through a **dedicated read-only cache**, and that the cache includes hardware filtering which can perform linear floating point interpolation as part of the read process. ”*

Motivation

- What to expect from a benchmark for GP computing?
  - Supports diverse applications with broad range of communication patterns
  - State-of-the-art algorithms
  - Input sets for testing different situations
- At the time of writing, most of the previous benchmarks focused on serial and parallel applications for **conventional GP-CPU architectures** rather than heterogeneous architectures.

Motivation

- **Compare two architectures** and identify inherent architectural advantages
- Decide **what hardware features should be included** in the limited area budgets
- Help compiler efforts to **port existing CPU languages/APIs to the GPU** by providing reference implementations
- Provides software developers with **exemplars** for different applications

Benchmark	Serial / Parallel	GPU / CPU	Purpose	Updated Since
SPEC	S	CPU	GP-CPU	Yes
EEMBC	S	CPU	GP-CPU	Yes+
SPLASH-2	S / P	CPU	GP-CPU	No*
PARSEC	S / P	CPU	GP-CPU	Yes
MineBench	S / P	GPU	Data Mining	Yes
MediaBench	S / P	GPU	Multimedia	Yes
ALP-Bench	S / P	GPU	Multimedia	No*
BioParallel	S / P	GPU	Biomedical	No+
Parboil	S / P	GPU	GP-GPU	Yes

The Rodinia Benchmark Suite

- Uses **Berkeley Dwarves as guidelines** for selecting benchmarks
- **Contains four applications and five kernels*
  - CPUs - Parallelized with **OpenMP**
  - GPUs - **CUDA**
  - *Similarity Score - Mars' MapReduce API*
- Workloads chosen to exhibit
  - Parallelism
  - Data access patterns
  - Data sharing characteristics

The Rodinia Benchmark Suite

TABLE I
RODINIA APPLICATIONS AND KERNELS (*DENOTES KERNEL).

Application / Kernel	Dwarf	Domain
K-means	Dense Linear Algebra	Data Mining
Needleman-Wunsch	Dynamic Programming	Bioinformatics
HotSpot*	Structured Grid	Physics Simulation
Back Propagation*	Unstructured Grid	Pattern Recognition
SRAD	Structured Grid	Image Processing
Leukocyte Tracking	Structured Grid	Medical Imaging
Breadth-First Search*	Graph Traversal	Graph Algorithms
Stream Cluster*	Dense Linear Algebra	Data Mining
Similarity Scores*	MapReduce	Web Mining

The Rodinia Benchmark Suite - Workloads

Leukocyte Tracking (LC)	Detect and track rolling leukocytes in video microscopy
S.R Anisotropic Diffusion (SRAD)	Removing speckles in an image without sacrificing features
HotSpot (HS)	Thermal simulation tool to estimate processor temperature
Back Propagation (BP)	Train neural networks by propagating error and updating weights
Needleman-Wunsch (NW)	DNA sequence alignment by score evaluation
K-Means (KM)	Clustering by finding centroids and adding points until convergence
Stream Cluster (SC)	Online clustering with a predetermined number of medians
Breadth First Search (BFS)	Traverse connected components in a graph
Similarity Score (SS)	Computing pairwise similarity between pairs of web documents

The Rodinia Benchmark Suite

- CUDA
  - GTX 280 GPU | 30 SM , 8 SPs : 240 SPs |16kB SMPB | 1GB
  - “SM contains 8 SP. These SMs only get one instruction at time which means that the 8 SPs all execute the same instruction. This is done through a warp where the 8 SPs spend 4 clock cycles executing a single instruction”[1]

- CUDA vs OpenMP
  - More fine-grained specification of tasks
  - Reductions must be handled manually

[1] https://devtalk.nvidia.com/default/topic/459248/difference-between-cuda-core-amp-streaming-multiprocessor/

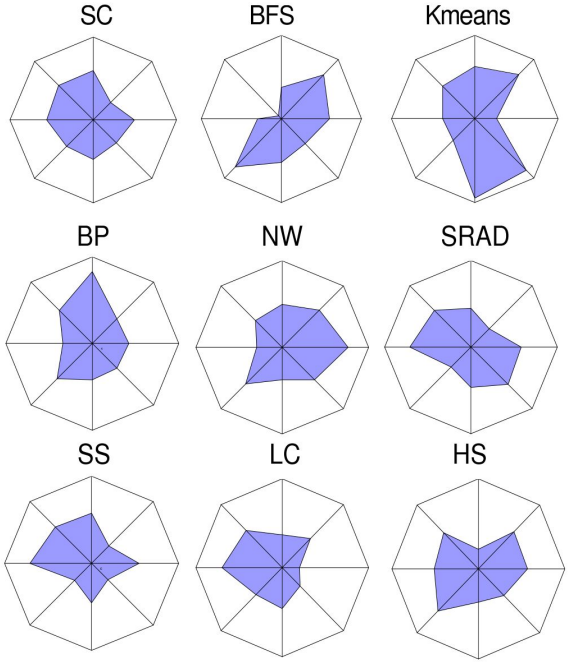
Methodology and Experiment Design

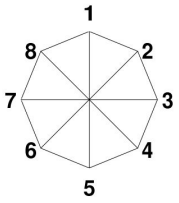
- Is the suite diverse enough? - **Diversity Analysis**
- Does the style of parallelization and optimization affect different target platforms? - **Parallelization and Speedup**
- To quantitatively evaluate the communication overhead between GPUs and CPUs - **Computation & Communication**
- Do synchronization primitives and strategies affect performance? - **Synchronization**
- Do both approaches (CPU , GPU) affect power-efficiency differently? - **Power Consumption**

Q5. Does this synchronization refer to synchronizing all the threads only in a block? Also, is this "overhead" occurring because in a kernel, the next task can't be started before all the threads in the previous task are done executing, hence the delay?

Diversity Analysis

- Microarchitecture Independent Workload Characterization(MICA) - A plugin for Linux PIN tool capable of characterizing the kernels independently from its running architecture by monitoring non-hardware features.
- Fairly accurate despite being compiler dependent. **Eg: SSE**
- Diversity of applications under consideration is shown in figure 1.
- GPU speedup - 5.5 to 80.8 times over single core and 1.6 to 26.3 times over quad-core CPUs excluding **I/O and initial setup**.
- LC, SRAD and HS - compute intensive.
- NW, BFS, KM and SC - memory bandwidth limited. **DS dependent.**
- SC, KM and SRAD mask memory latency with data parallelism.





1. Prob register dependence dist. ⩽ 16
2. Branch predictability
3. Percentage arithmetic operations
4. Data working set size (32-byte block)
5. Prob. local store stride = 0
6. Prob. global store stride ⩽ 8
7. Prob. local store stride ⩽ 8
8. Prob. local store stride ⩽ 4096

Fig. 1. Kiviat diagrams representing the eight microarchitecture-independent characteristics of each benchmark.

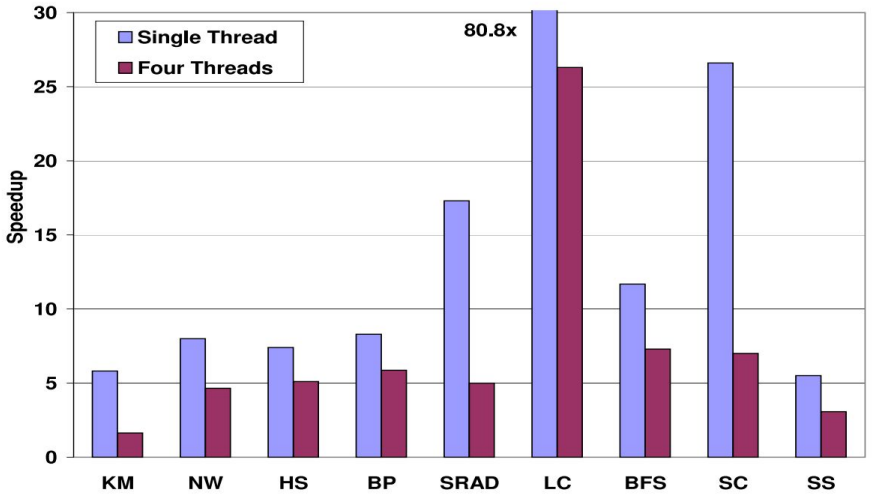


Fig. 2. The speedup of the GPU implementations over the equivalent single- and four-threaded CPU implementations. The execution time for calculating the speedup is measured on the CPU and GPU for the core part of the computation, excluding the I/O and initial setup. Figure 4 gives a detailed breakdown of each CUDA implementation's runtime.

Parallelization and Optimization

- After performance, GPU optimizations: CPU-GPU communications & memory coalescing. Neighbouring threads access sequential memory. **Eg: BFS**
- Caching is a good for large read-only data structures.
- If sufficient parallelism is available, then gains from efficient thread-bandwidth usage can mask memory access latencies.

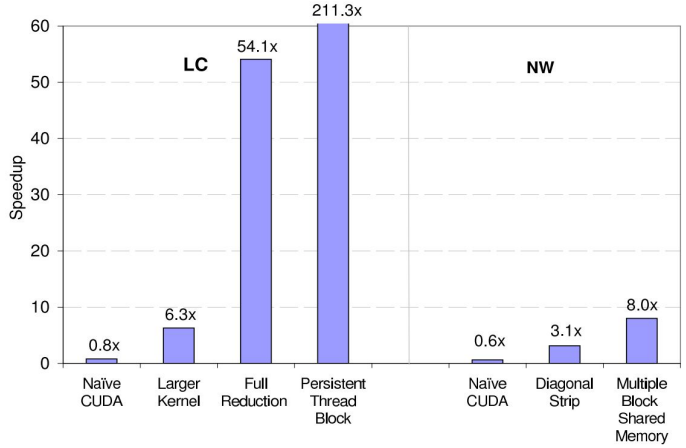
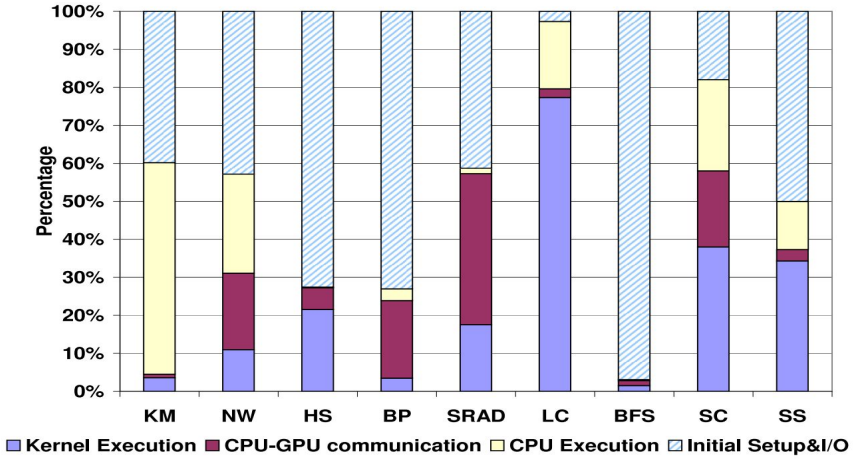


Fig. 3. Incremental performance improvement from adding optimizations

Computation and Communication

- Programs with largest problem sizes have highest miss-rates. **True!**
- Amdahl's law! - gives the upper bound of parallelism based performance
- Disjoint CPU-GPU address spaces needs translation.
- Moving work to GPU despite higher CPU efficiency can be beneficial if it reduces CPU-GPU communication. **(Fig *)**



Synchronization (CUDA)

- **Intra-block synchronization**
  - Use `__syncthreads()` to synchronize within a thread block
- **Global (inter-block) synchronization**
  - Multiple kernel launches are required
  - This adds significant overhead
- **Atomic instructions**
  - These have likely improved over time, but authors note bandwidth was poor circa 2009
  - We weren't able to find a recent paper to update this assessment
- **Conclusion?**
  - Keep synchronization and communication **local to thread blocks whenever possible**

Table II (CUDA Synchronization)

TABLE II
APPLICATION INFORMATION. KN = KERNEL N; C = CONSTANT MEMORY; CA = COALESCED MEMORY ACCESSES; T = TEXTURE MEMORY; S = SHARED MEMORY.

	KM	NW	HS	BP	SRAD	LC	BFS	SC	SS
Registers Per Thread	K1:5 K2:12	K1:21 K2:21	K1:25	K1:8 K2:12	K1:10 K2:12	K1:14 K2:12 K3:51	K1:7 K2:4	K1:7	K1,4,6,7,9-14:6 K2:5 K3:10 K5:13 K8:7
Shared Memory	K1:12 K2:2096	K1:2228 K2:2228	K1:4872	K1:2216 K2:48	K1:6196 K2:5176	K1:32 K2:40 K3:14636	K1:44 K2:36	K1:80	K1:60 K2-4:48 K5,8:40 K9:12 K6,7,10,11:32 K12-14:36
Threads Per Block	128/256	16	256	512	256	128/256	512	512	128
Kernels	2	2	1	2	2	3	2	1	14
Barriers	6	70	3	5	9	7	0	1	15
Lines of Code[2]	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 data points	219×640 pixels/frame	10^6 nodes	65536 points 256 dimensions	256 points 128 features
CPU Execution Time[3]	20.9 s	395.1 ms	3.6 s	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
Parallel Overhead (%)	14.8	32.4	35.7	33.8	4.1	2.2	29.8	2.6	27.7

Back to Figure 2

- **NW and SS** have among the **worst speedups** compared to CPU implementations
  - This could partially be attributed to the relatively large amount of synchronization needed
  - But there are of course other factors at play

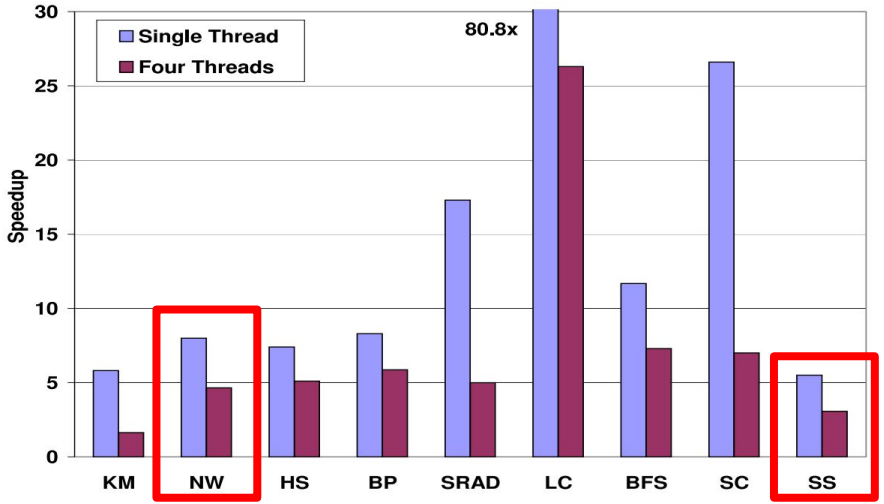


Fig. 2. The speedup of the GPU implementations over the equivalent single- and four-threaded CPU implementations. The execution time for calculating the speedup is measured on the CPU and GPU for the core part of the computation, excluding the I/O and initial setup. Figure 4 gives a detailed breakdown of each CUDA implementation's runtime.

Synchronization (OpenMP)

- Parallel constructs have implicit barriers
  - “Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, [...]”[1]
- Programmers also have a rich set of synchronization features
  - e.g. ATOMIC directive
  - `#pragma omp atomic`
    `expression`
  - Parameters: `expression` - The statement containing the lvalue whose memory location you want to protect against multiple writes.

[1] http://www.openmp.org/wp-content/uploads/cspec20.pdf

Table II (OpenMP)

TABLE II
APPLICATION INFORMATION. KN = KERNEL N; C = CONSTANT MEMORY; CA = COALESCED MEMORY ACCESSES; T = TEXTURE MEMORY; S = SHARED MEMORY.

	KM	NW	HS	BP	SRAD	LC	BFS	SC	SS
Registers Per Thread	K1:5 K2:12	K1:21 K2:21	K1:25	K1:8 K2:12	K1:10 K2:12	K1:14 K2:12 K3:51	K1:7 K2:4	K1:7	K1,4,6,7,9-14:6 K2:5 K3:10 K5:13 K8:7
Shared Memory	K1:12 K2:2096	K1:2228 K2:2228	K1:4872	K1:2216 K2:48	K1:6196 K2:5176	K1:32 K2:40 K3:14636	K1:44 K2:36	K1:80	K1:60 K2-4:48 K5,8:40 K9:12 K6,7,10,11:32 K12-14:36
Threads Per Block	128/256	16	256	512	256	128/256	512	512	128
Kernels	2	2	1	2	2	3	2	1	14
Barriers	6	70	3	5	9	7	0	1	15
Lines of Code[2]	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								points nensions	256 points 128 features
CPU Execution Time[3]								1.0 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
Parallel Overhead (%)	14.8	32.4	35.7	33.8	4.1	2.2	29.8	2.6	27.7

Parallel Overhead = $(T_p - T_s/p)$, where T_p is execution time on p processors and T_s is sequential execution time

Back to Figure 2

- **SRAD and LC** are helped dramatically by parallelization
  - The authors attribute this to highly **independent computations** within the SRAD and LC kernels
  - But there are of course other factors at play

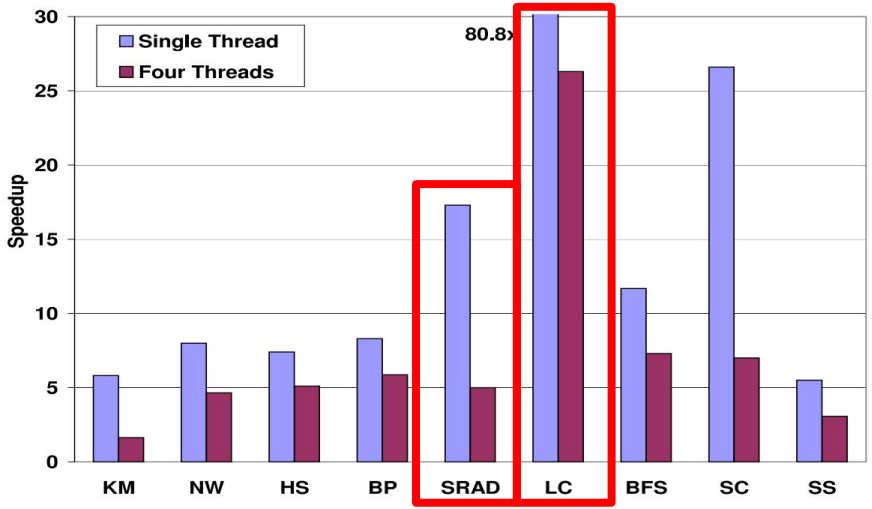


Fig. 2. The speedup of the GPU implementations over the equivalent single- and four-threaded CPU implementations. The execution time for calculating the speedup is measured on the CPU and GPU for the core part of the computation, excluding the I/O and initial setup. Figure 4 gives a detailed breakdown of each CUDA implementation's runtime.

Power Consumption

- Power benchmarks were done **for each kernel's three versions:** GPU, single CPU core, and four CPU cores
- **Extra power dissipation**
  - $\text{Power}_{\text{idle}} - \text{Power}_{\text{kernel}}$
  - The authors' system idles at **186 W**, which includes the idle power of the GPU
- The authors seem to be using some notion of **average power**, although in our opinion, an **energy measurement** might've been more interesting

Figure 5 (Some interesting results)

- In **BP, SS, and KM**, the **GPU consumes less power** than the four CPU cores.
  - **Why?** The answer differs for each kernel, but here are some contributing factors: **KM** exploits special GPU memory, **KM and BP** don't use much shared memory, etc.

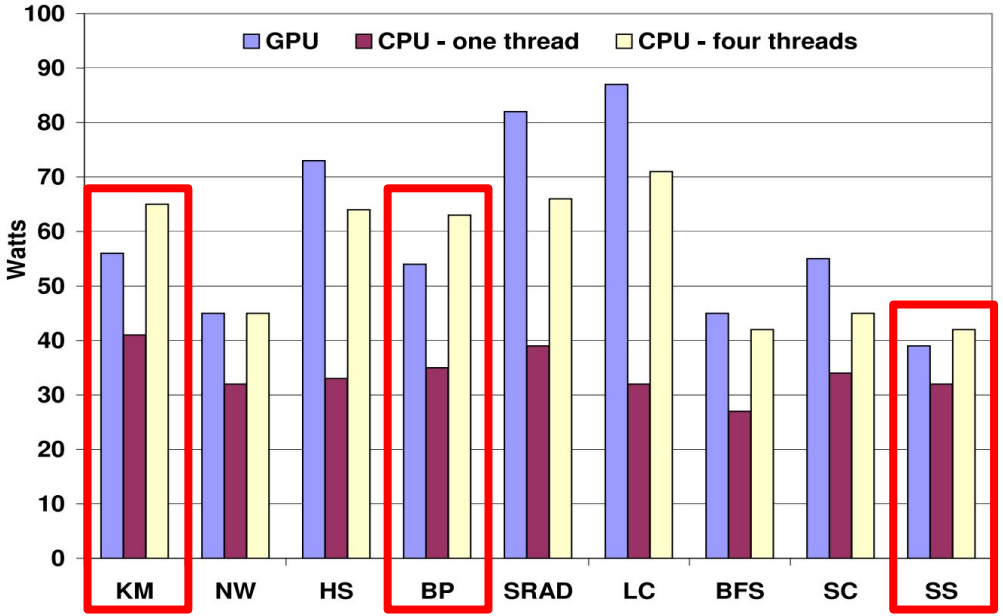


Fig. 5. Extra power dissipation of each benchmark implementation in comparison to the system's idle power (186 W).

Figure 5 (Some interesting results)

- In **BP, SS, and KM**, the **GPU consumes less power** than the four CPU cores.
- For **NW**, the **CPU and the GPU consume similar amounts of power**

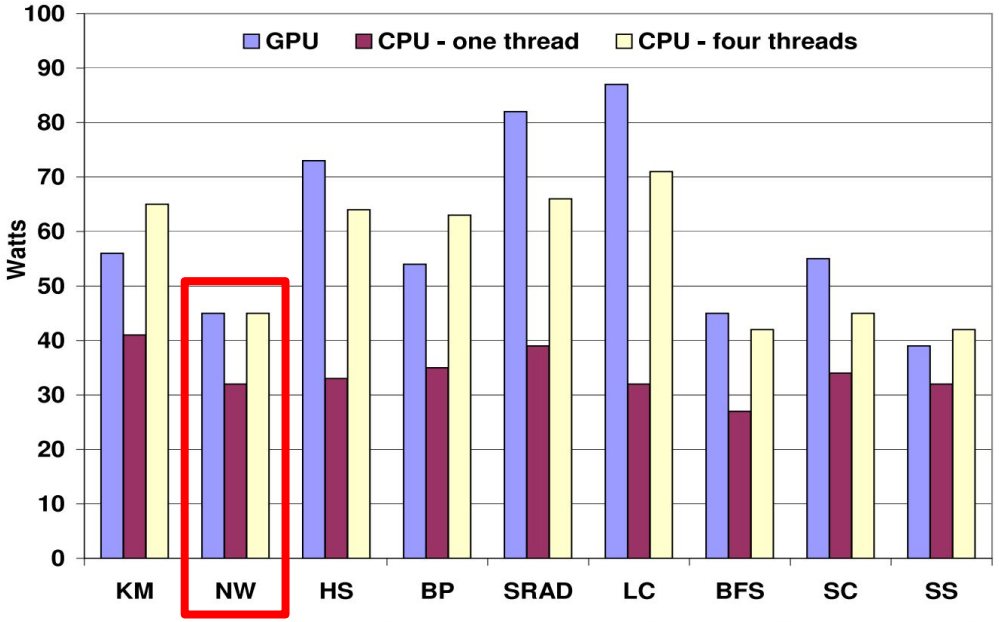


Fig. 5. Extra power dissipation of each benchmark implementation in comparison to the system's idle power (186 W).

Figure 5 (Some interesting results)

- **Speedup per watt**
  - It's **mostly more efficient** to run on GPU
  - e.g., SRAD dissipates 24% more power on GPU than on four-core CPU, but speedup over multicore is 5.0
  - **NW** efficiency is **roughly the same** in GPU and CPU
  - Why? NW presents little parallelism within its diagonal strip access pattern

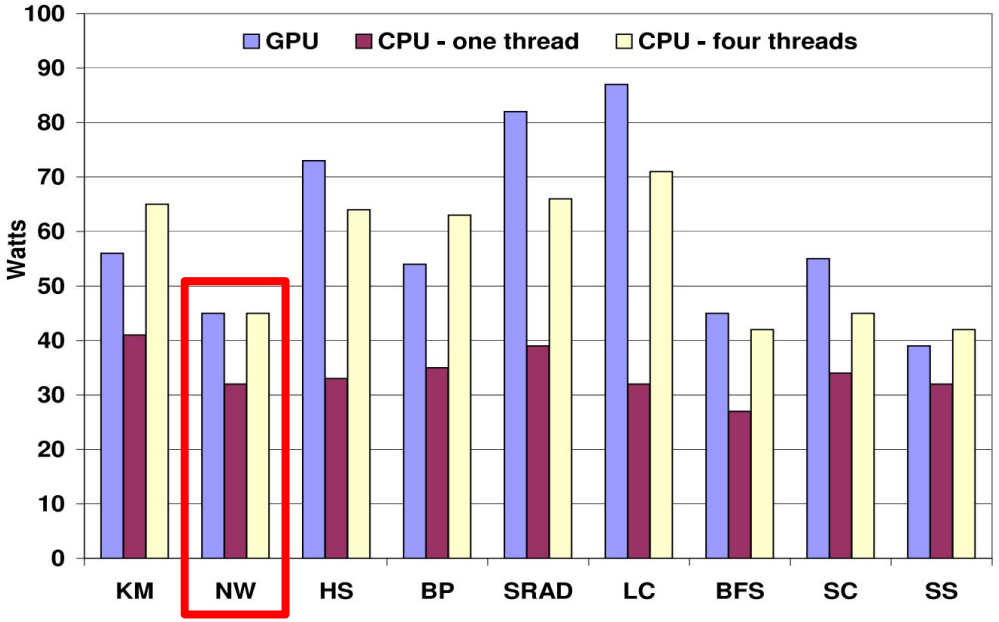
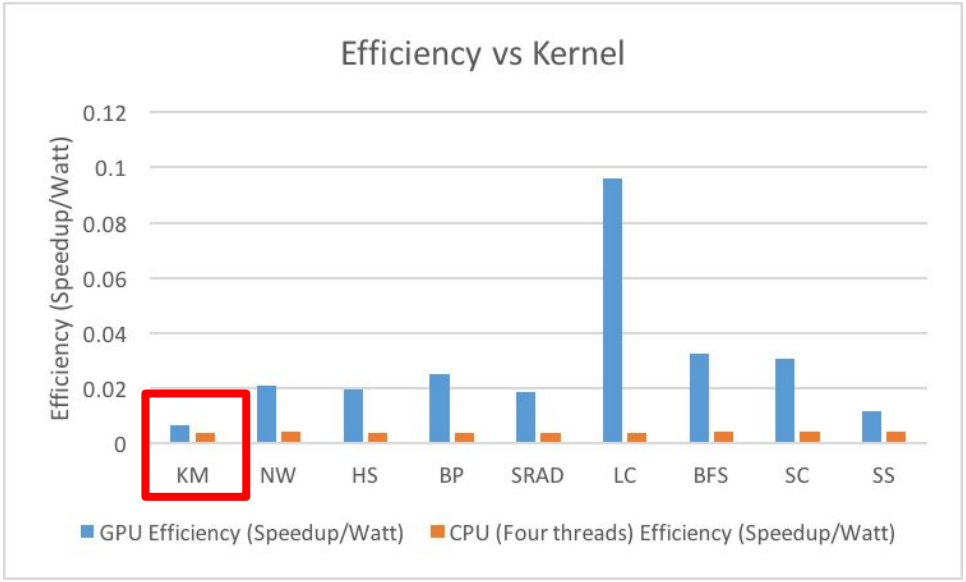


Fig. 5. Extra power dissipation of each benchmark implementation in comparison to the system's idle power (186 W).

A small side note...

- **Can someone explain this to me?**
  - Smallest difference in efficiency occurs when running KM (difference of 0.0027)
  - **Did the authors mean KM?**
  - This seems to make sense, if you consider the speedup for KM was only 1.6x!



Can you briefly describe what causes the GPU to require more energy for the execution of their workloads?

Can you briefly describe what causes the GPU to require more energy for the execution of their workloads?

In general, GPUs are executing the workload faster and more efficiently (see previous speedup per watt slide). Like any trade off, we don't get this speedup for free: we're paying for the speedup with additional power dissipation.

Discussion (CUDA)

- **Data structure mapping**
  - Programmers must map their application's data structures to the CUDA domain (CUDA loves matrices)
- **Global memory fence**
  - The lack of a global memory fence forces programmers to launch multiple kernel to synchronize (costly overhead)
- **Memory transfer**
  - Disjoint memory spaces adds to overhead, but CUDA does provide a streaming interface
    - Overlaps computations with memory transfers
- **Offloading Decision**
  - It isn't always intuitive what to run on a GPU
- **Resource considerations**
  - Per-thread storage is tiny in the register file, texture cache, and shared memory

In the "Memory Transfer" section, it’s stated that batch kernel calls can work efficiently only if "there is no CPU code between GPU kernel calls, and there are multiple independent streams of work." How could intermediate CPU code execution effect the kernel calls since they are being executed in different hardware?

In the "Memory Transfer" section, it's stated that batch kernel calls can work efficiently only if "there is no CPU code between GPU kernel calls, and there are multiple independent streams of work." How could intermediate CPU code execution effect the kernel calls since they are being executed in different hardware?

We think the key phrase here is “work efficiently”... You can introduce intermediate CPU code, but this wouldn’t always hide the memory transfer as efficiently.

Further reading: https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Discussion (OpenMP)

- Compiler directives, library routines, etc. give programmers control over parallelism
- Programmers still must determine what to parallelize
- Programmers still must avoid data races

Discussion (OpenCL)

- OpenCL **platform and memory models** are very similar to CUDA
- If **Rodinia applications** were implemented in **OpenCL**, many of the same lessons and optimizations could be applied
  - Today, we have good OpenCL support in Rodinia

Applications	Dwarves	Domains	Parallel Model	Incre. Ver.
Leukocyte	Structured Grid	Medical Imaging	CUDA, OMP, OCL	✔
Heart Wall	Structured Grid	Medical Imaging	CUDA, OMP, OCL	
MUMmerGPU	Graph Traversal	Bioinformatics	CUDA, OMP	
CFD Solver[1]	Unstructured Grid	Fluid Dynamics	CUDA, OMP, OCL	
LU Decomposition	Dense Linear Algebra	Linear Algebra	CUDA, OMP, OCL	✔
HotSpot	Structured Grid	Physics Simulation	CUDA, OMP, OCL	
Back Propagation	Unstructured Grid	Pattern Recognition	CUDA, OMP, OCL	
Needleman-Wunsch	Dynamic Programming	Bioinformatics	CUDA, OMP, OCL	✔
Kmeans	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL	
Breadth-First Search[1]	Graph Traversal	Graph Algorithms	CUDA, OMP, OCL	
SRAD	Structured Grid	Image Processing	CUDA, OMP, OCL	✔
Streamcluster[1]	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL	
Particle Filter	Structured Grid	Medical Imaging	CUDA, OMP, OCL	
PathFinder	Dynamic Programming	Grid Traversal	CUDA, OMP, OCL	
Gaussian Elimination	Dense Linear Algebra	Linear Algebra	CUDA, OCL	
k-Nearest Neighbors	Dense Linear Algebra	Data Mining	CUDA, OMP, OCL	
LavaMD[2]	N-Body	Molecular Dynamics	CUDA, OMP, OCL	
Myocyte	Structured Grid	Biological Simulation	CUDA, OMP, OCL	
B+ Tree	Graph Traversal	Search	CUDA, OMP, OCL	
GPUDWT	Spectral Method	Image/Video Compression	CUDA, OCL	
Hybrid Sort	Sorting	Sorting Algorithms	CUDA, OCL	
Hotspot3D	Structured Grid	Physics Simulation	CUDA, OCL, OMP	Hotspot for 3D IC
Huffman	Finite State Machine	Lossless data compressio	CUDA, OCL	

Discussion (PGI generated GPU code)

- Paper recommendation: **Directive-Based Compilers for GPUs**
  - Swapnil Ghike, Ruben Gran, Maria J. Garzaran, and David Padua, 2015
  - “In terms of performance, the versions compiled Cray performed faster than the ones of PGI compiler for 8 out of 15 Rodinia benchmarks. In comparison to fine-tuned CUDA versions, 6 out of 15 heterogeneous versions ran over the 85% of the CUDA performance. This shows the potential of these heterogeneous directives-based compilers to produce efficient code and at the same time increase programmer productivity.”