CMPE 214
GPU Architecture & Programming

Lecture 3. Profiling and Simulation (2)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

GPU Performance Metrics

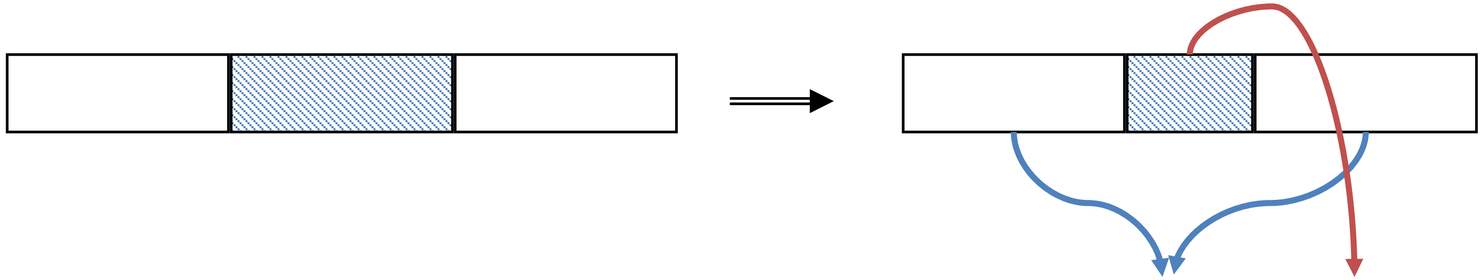
- **Run time**
  - Real execution
  - Simulation
  - Profiling
- **Throughput**
  - **IPC: instruction per cycle**
- **Occupancy = active warps / max warps**
- **Memory metrics:**
  - **L1/L2 MR**
  - **Bandwidth Utilization: BW_UTIL = cycles in read and write / total dram cycles**
  - **Row buffer locality: RBL = requests / row activation**
- **Amdahl's law: More code portions should be parallelizable**

Multicore Performance: Amdahl's Law

- **Speedup due to enhancement E:**

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- **Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected:**



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times ((1-F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

Example 1: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- **Consider an enhancement that runs 20 times faster but is only usable 25% of the time**

  Speedup w/ E = 1/(.75 + .25/20) = 1.31

- **What if its usable only 15% of the time?**

  Speedup w/ E = 1/(.85 + .15/20) = 1.17

- **Amdahl's Law tells us that to achieve linear speedup with 100 cores, none of the original computation can be scalar!**

- **To get a speedup of 90 from 100 cores, the percentage of the original program that could be scalar would have to be 0.1% or less**

  Speedup w/ E = 1/(.001 + .999/100) = 90.99

Example 2: Amdahl's Law

Speedup w/ E = 1 / ((1-F) + F/S)

- **Consider summing 10 scalar variables and two 10 by 10 matrices on 10 cores**

  Speedup w/ E = 1/(.091 + .909/10) = 1/0.1819 = 5.5

- **What if there are 100 cores?**

  Speedup w/ E = 1/(.091 + .909/100) = 1/0.10009 = 10.0

- **What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 cores?**

  Speedup w/ E = 1/(.001 + .999/10) = 1/0.1009 = 9.9

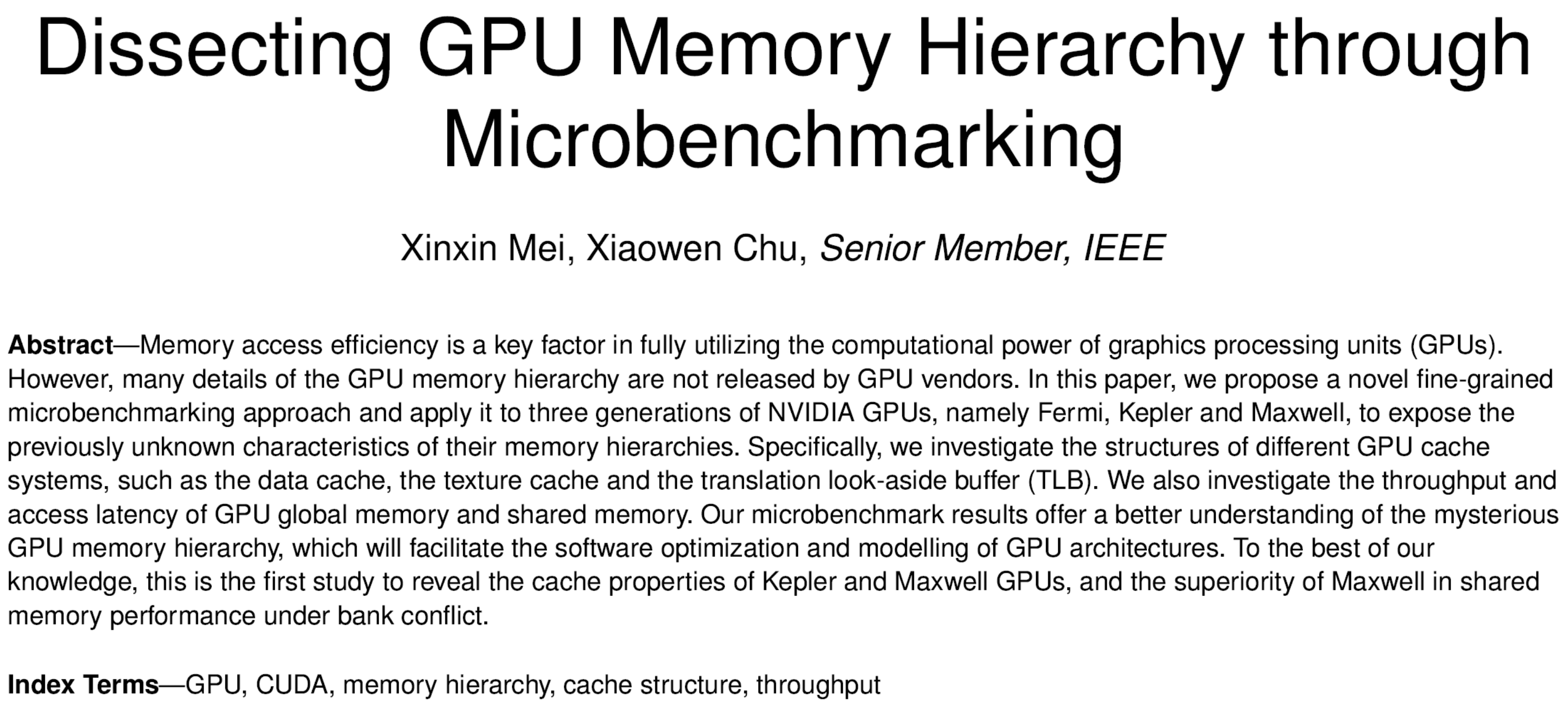
- **What if there are 100 cores?**

  Speedup w/ E = 1/(.001 + .999/100) = 1/0.01099 = 91

Time Profiling: P-chasing Method

```
__device__ void P_chasing1(int *A, long long int iterations, int starting_index){

        int j = starting_index;

        long long int start_time = 0;//////clock
        long long int end_time = 0;//////clock

        start_time = clock64();//////clock

        for (int it = 0; it < iterations; it++){
                j = A[j];
        }

        end_time=clock64();//////clock

        long long int total_time = end_time - start_time;//////clock
}
```

State-of-the-art



Dissecting GPU Memory Hierarchy through Microbenchmarking

Xinxin Mei, Xiaowen Chu, *Senior Member, IEEE*

Abstract—Memory access efficiency is a key factor in fully utilizing the computational power of graphics processing units (GPUs). However, many details of the GPU memory hierarchy are not released by GPU vendors. In this paper, we propose a novel fine-grained microbenchmarking approach and apply it to three generations of NVIDIA GPUs, namely Fermi, Kepler and Maxwell, to expose the previously unknown characteristics of their memory hierarchies. Specifically, we investigate the structures of different GPU cache systems, such as the data cache, the texture cache and the translation look-aside buffer (TLB). We also investigate the throughput and access latency of GPU global memory and shared memory. Our microbenchmark results offer a better understanding of the mysterious GPU memory hierarchy, which will facilitate the software optimization and modelling of GPU architectures. To the best of our knowledge, this is the first study to reveal the cache properties of Kepler and Maxwell GPUs, and the superiority of Maxwell in shared memory performance under bank conflict.

Index Terms—GPU, CUDA, memory hierarchy, cache structure, throughput

- **Dissecting GPU Memory Hierarchy through Microbenchmarking**
  - Fine-grained p-chasing

https://arxiv.org/pdf/1509.02308&ved=0ahUKEwifl_P9rt7LAhXBVxoKHRsxDIYQFgg_MAk&usg=AFQjCNGchkZRzkueGqHEz78QnmclVCSXvg&sig2=IdzxfrzQgNv8yq7e1mkeVg

Fine-grained P-chasing

```
__device__ void P_chasing1(int *A, long long int iterations, int starting_index){

        __shared__ long long int s_tvalue[1024 * 4];/////must be enough to contain the number of iterations.
        __shared__ int s_index[1024 * 4];
        int j = starting_index;

        long long int start_time = 0;//////clock
        long long int end_time = 0;//////clock
        long long int time_interval = 0;//////clock

        for (int it = 0; it < iterations; it++){
                start_time = clock64();//////clock
                j = A[j];

                s_index[it] = j;
                end_time=clock64();//////clock
                s_tvalue[it] = end_time - start_time;
        }
}
```

One Step Further: Using PTX

- **You can write PTX code directly in CUDA**
  - Format: asm("template-string" : "constraint"(output) : "constraint"(input));

- **Example: asm("add.s32 %0, %1, %2;" : "=r"(i) : "r"(j), "r"(k));**
  - add.s32 i, j, k;

- **Constraint:**
  - The "=" modifier specifies that the register is written to.
  - The "+" modifier specifies the register is both read and written.
  - "h" = .u16 reg
  - "r" = .u32 reg
  - "l" = .u64 reg
  - "f" = .f32 reg
  - "d" = .f64 reg

- Nvidia doc: https://docs.nvidia.com/cuda/inline-ptx-assembly/index.html

Fine-grained P-chasing with ASM

```
…
        asm(".reg .u64 t1;\n\t"
        ".reg .u64 t2;\n\t");

        for (long long int it = 0; it < iterations; it++){

                asm("mul.wide.u32  t1, %2, %4;\n\t"
                "add.u64        t2, t1, %3;\n\t"
                "mov.u64        %0, %clock64;\n\t"
                "ld.global.u32          %1, [t2];\n\t"
                : "=l"(start_time), "=r"(j) : "r"(j), "l"(A), "r"(4));

                s_index[it] = j;

                asm volatile ("mov.u64 %0, %clock64;": "=l"(end_time));

                time_interval = end_time - start_time;
                s_tvalue[it] = time_interval;
        }
…
```

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY