# [A3] Pollack's Rule as a Justification for Heterogeneous Computing

# Module overview and context

- Students assumed to have seen argument for multicore (Moore's law and power limits on frequency scaling)
- Should lead into concrete instance of heterogeneous architecture (SSE instructions, ARM Thumb mode, CUDA, …)
- Resources provided: Slides and HW/exam exercises

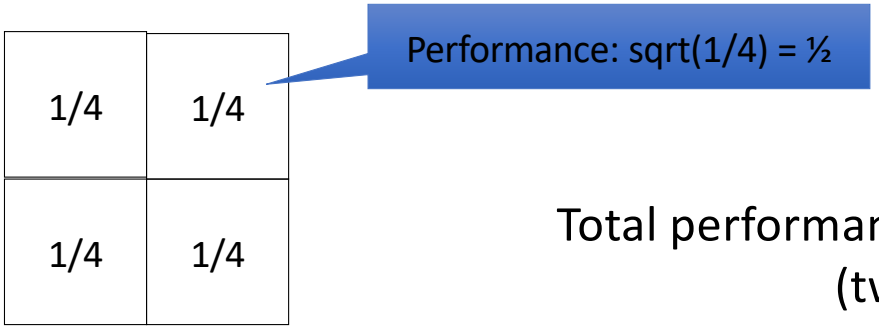
# Pollack's rule

- The performance of a processing core is proportional to the square root of its area

# Pollack's rule

- The performance of a processing core is proportional to the square root of its area

If a single core is replaced by 4 cores, each ¼ as large, what is the expected peak performance of the entire system? (i.e. the performance assuming all 4 could be kept perfectly busy)




Total performance: 4 × ½ = 2
(twice as much)

How does the running time change when a single core is replaced with 4 cores if only half the program can be parallelized?

- Parallel part:

  ½ the work / 2 the performance = ¼

- Serial part:

  ½ the work / ½ the performance = 1

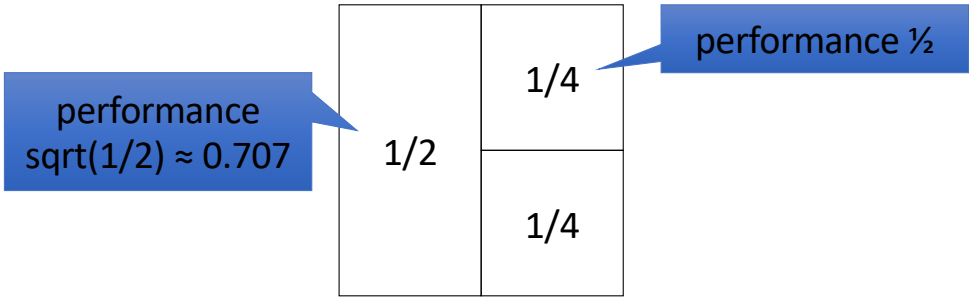
Total time: 1.25 times as long

# Factor by which running time changes for different programs

| % of program that is parallelizable | 75% | 90% | 95% |
|---|---|---|---|
| 1 core | 1 | 1 | 1 |
| 4 cores | 0.875 | 0.65 | 0.575 |
| 9 cores | 1 | 0.6 | 0.467 |
| 16 cores | 1.1875 | 0.625 | 0.438 |
| 25 cores | 1.4 | 0.68 | 0.44 |
| 36 cores | 1.625 | 0.75 | 0.458 |

As the number of cores increases, highly parallelizable programs have improved performance, but less parallelizable programs suffer

# What about unequal core sizes?




By what factor does the peak performance of this system differ from a single core?

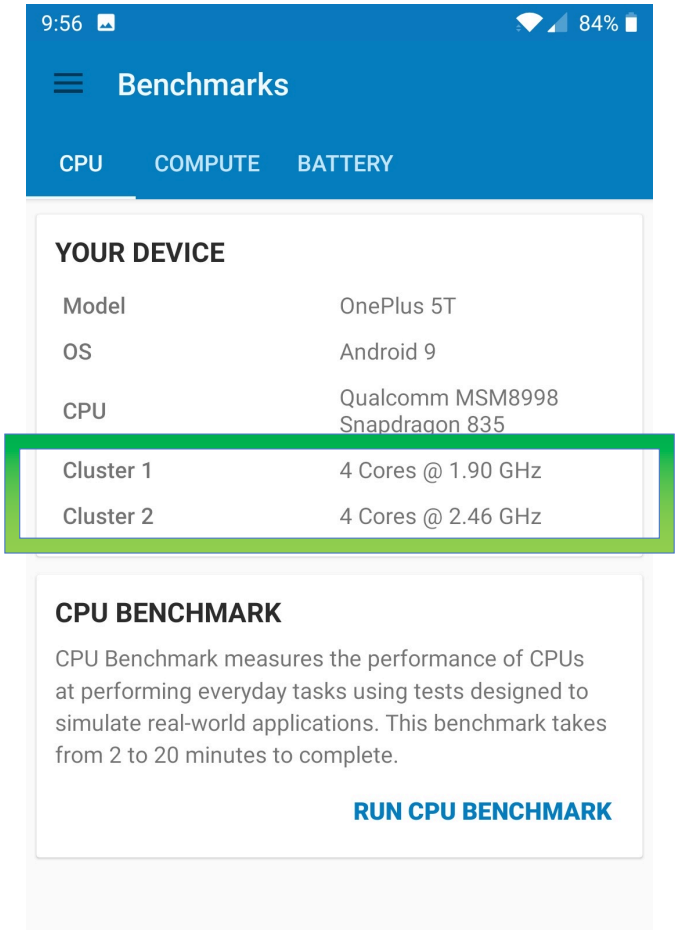
0.707 + 2 ✕ 0.5 = 1.707 times as much

# Factor by which running time changes for different programs

| % of program that is parallelizable | 50% | 75% | 90% |
|---|---|---|---|
| 4 equal cores | 1.25 | 0.88 | 0.65 |
| Half-sized + 2 quarter-sized cores | 1.00 | 0.79 | 0.66 |

Having different sized cores improves performance on less parallelizable programs at small cost on more highly parallelizable ones

# Heterogeneity on a cell phone

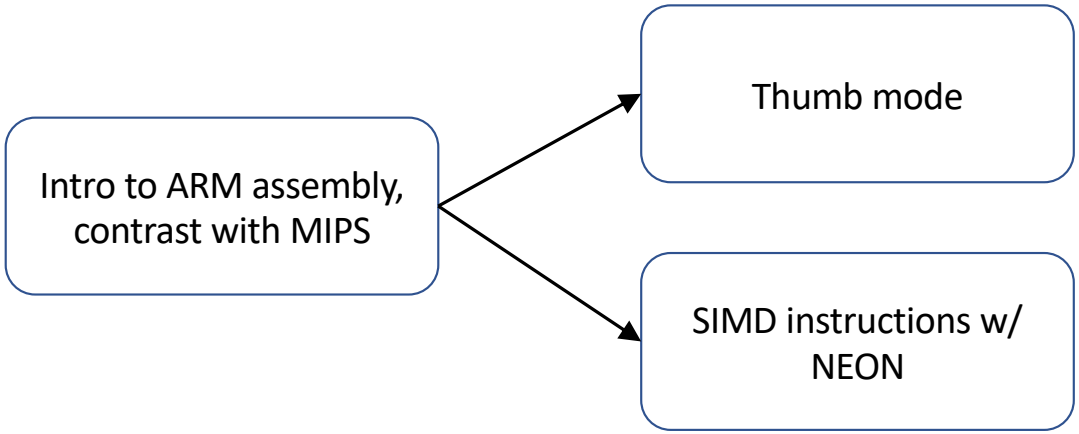



8 cores, 2 levels of performance

# [C1] Introduction to ARM

# “Module” contents

- Really three parts, each built around a lab
- All use/teach ARM assembly language, probably running it on a Raspberry Pi



# Intro to ARM Assembly

- Assumes students have learned MIPS assembly (in this class)
- Show that “assembly” is not one language by contrasting with a related (still RISC) assembly language
- Run on real hardware
- 1-2 lectures and a lab

# ARM differences from MIPS

- Addressing modes that increment as part of load/store

- Conditional instructions:

```
CMP R5, #0          @set status register
SUBNE R5, R5, #1    @R5=R5-1 if R5 !=0
MOVEQ R5, #40       @R5=40 if R5 == 0
```

- Function stack managed with push and pop operations

# Thumb mode

- Show heterogeneity within ARM assembly

- Generally:
  - ARM code: all instructions are 32-bit
  - Thumb-1 code: all instructions are 16-bit  (often simply called “Thumb”)
  - Thumb-2 code: mixes 16 and 32-bit instructions

- Can move between ARM code and Thumb code within the same executable

# The tradeoff

- In Thumb:
  - Have limited access to some of the registers
  - Have subset of instructions and limited addressing modes
  - No floating point operations

- A 2002 paper by Krishnaswamy and Gupta found Thumb-1 code to be roughly 70% the size of ARM code on average and up to 30% slower
- Thumb mode can also reduce power consumption

# Thumb module

- Lecture + lab
  - Lab has them compile code for different modes
  - Observe code size, measure performance, compare code itself

# SIMD: Single Instruction, Multiple Data

- The idea is that a single CPU instruction is executed but multiple instances of that operation run on different pieces of data

- Consider adding two arrays elementwise:

```
for(int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

The additions are independent so we'd like to execute them in parallel

# ARM NEON data processing

- NEON coprocessor has 32 64-bit NEON registers that can be paired up
- These registers can be treated as vectors
- NEON instructions include a suffix to indicate the type of data
- **`VADD.I16 Q0, Q1, Q2`**
  - Each 128-bit Q-register is being treated as 8 16-bit signed integers. The addition will take place like this:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `Q1[0]` | `Q1[1]` | `Q1[2]` | `Q1[3]` | `Q1[4]` | `Q1[5]` | `Q1[6]` | `Q1[7]` |
| + | + | + | + | + | + | + | + |
| `Q2[0]` | `Q2[1]` | `Q2[2]` | `Q2[3]` | `Q2[4]` | `Q2[5]` | `Q2[6]` | `Q2[7]` |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| `Q0[0]` | `Q0[1]` | `Q0[2]` | `Q0[3]` | `Q0[4]` | `Q0[5]` | `Q0[6]` | `Q0[7]` |

# CUDA Modules ([D1] and [C2])

# CUDA

- Using compute capability of Nvidia graphics processing unit for general purpose computation (GPGPU)
- Model has many compute cores, but limits on their use
  - SIMD (Single Instruction, Multiple Data) programming
- Requires explicit data transfer

# “Hello World”

```
#include <stdio.h>
__global__ void hello() {
        int id = threadIdx.x + blockIdx.x * blockDim.x;
        printf("Hello from thread %d (%d of block %d)\n", id, threadIdx.x, blockIdx.x);
}

int main() {
        hello<<<5,4>>>();                       //launch 5 blocks of 4 threads each
        cudaDeviceSynchronize();                //make sure kernel completes
}
```

# Google Colab

- Cloud programming environment based on Jupyter notebooks
- Allows use of GPUs and running CUDA
- Requires programming in notebook cells; limited ability to save

# Module 1: Introduction to CUDA

https://github.com/TeachingUndergradsCHC/modules/tree/master/Programming/cuda

- Goal: Show students the potential and some of the complexity of CUDA (or GPGPU programming more generally)

- Where to use it: First exposure to CUDA
  - Requires C programming, familiarity with concepts of parallelism

- Lecture followed by a lab

# Module 1: Introduction to CUDA

https://github.com/TeachingUndergradsCHC/modules/tree/master/Programming/cuda

- Introduce GPGPU model
  - Need to transfer data
  - SIMD computation
- “Hello World” example
- Elementwise vector addition
- Why have multiple blocks and non-trivial blocks?
- Storing a 2D array in one dimension

# Lab: Image processing

Remove red (given code)

Convert to grayscale

Blur (average colors w/ neighbors)

# Module 2: GPU memory hierarchy

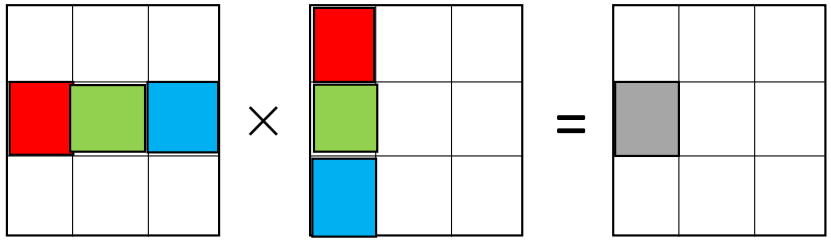
https://github.com/TeachingUndergradsCHC/modules/tree/master/Architecture/gpu_memory_hierarchy

- Goals:
  - Show that CUDA has different kinds of memory and one kind of optimization
  - Reinforce importance of memory locality
- Where to use it:
  - Requires prior familiarity with CUDA (e.g. from previous module)
  - Following discussion of caching and memory hierarchy
- Lecture and then lab/HW (more closely tied)

# Module 2: GPU memory hierarchy

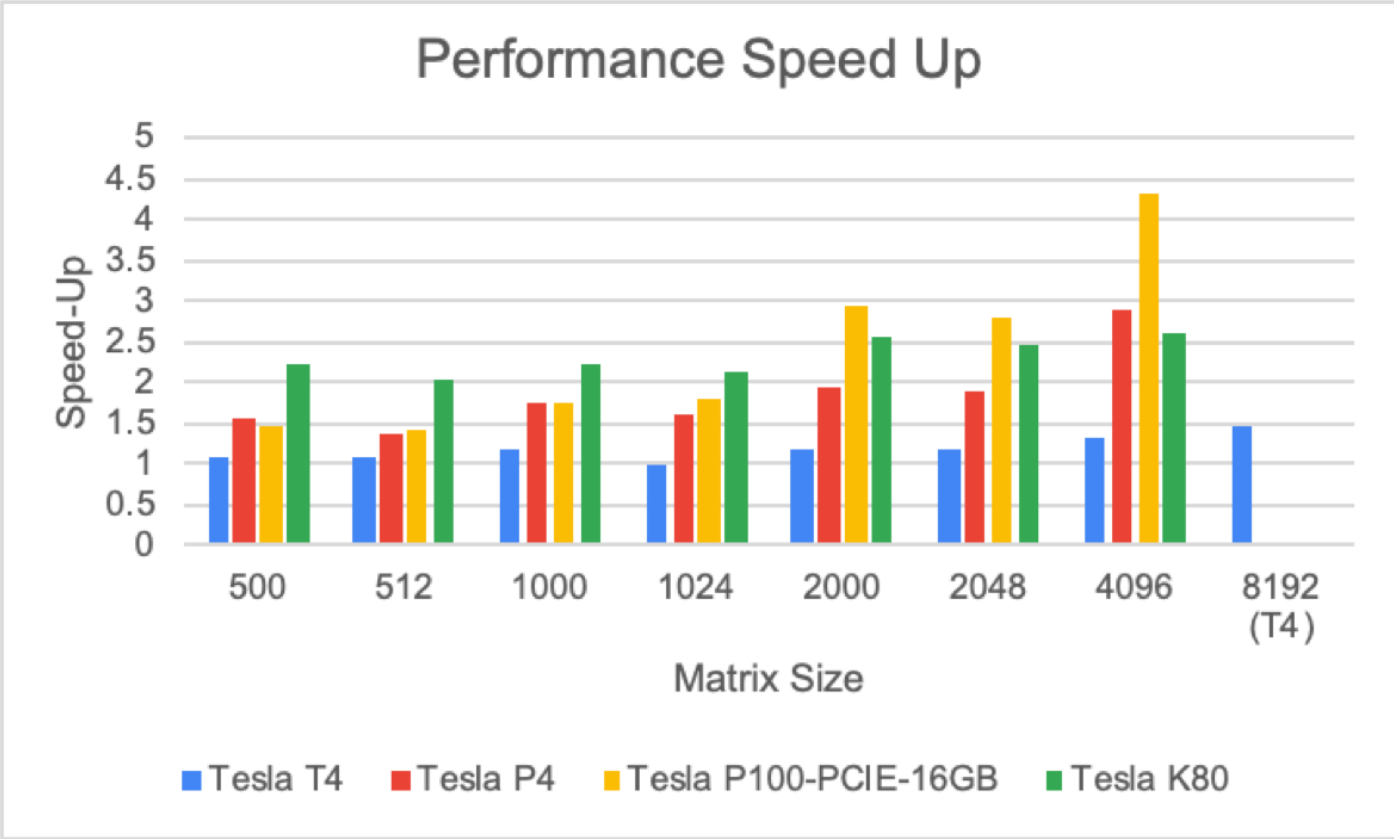
https://github.com/TeachingUndergradsCHC/modules/tree/master/Architecture/gpu_memory_hierarchy

- Tiling to reduce memory demands of matrix multiplication
- Using shared memory as programmer-managed cache



- Each thread is responsible for one location in a tile
- To generate gray tile of output, loop over row/col tiles
  - Work together to load 1 tile of row and 1 of column (e.g. the red tiles)
  - Calculate the contribution of those tiles to the output

# Performance results




Speed-up = Time for untiled version / time for tiled version

# Writing the tiled kernel

```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
        …
}
```

Step 1: Allocate the cache to store a tile of each matrix

# Writing the tiled kernel

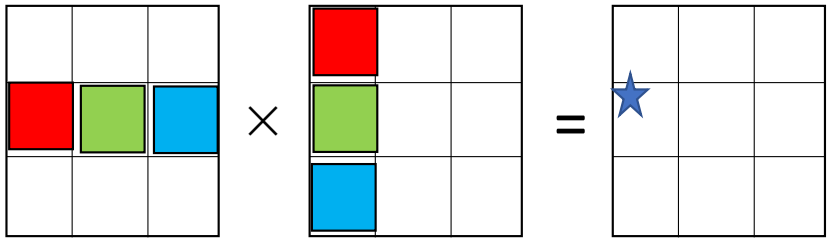
```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        …
        int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
        int col = blockIdx.x * TILE_WIDTH + threadIdx. x;

        int num_tiles = (width+TILE_WIDTH-1)/TILE_WIDTH;
        …
}
```

Step 2: Figure out the index for which this thread is responsible

(Block and tile have the same width)

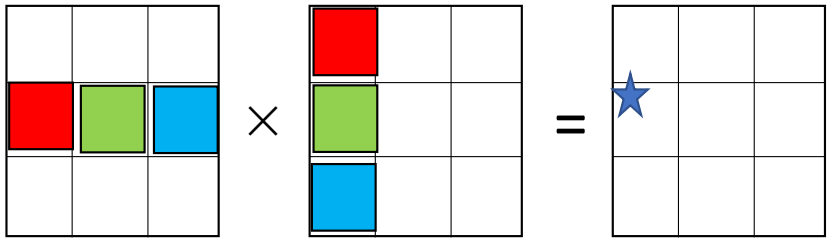
Figure out the width of the matrix in tiles

# Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {

}
```

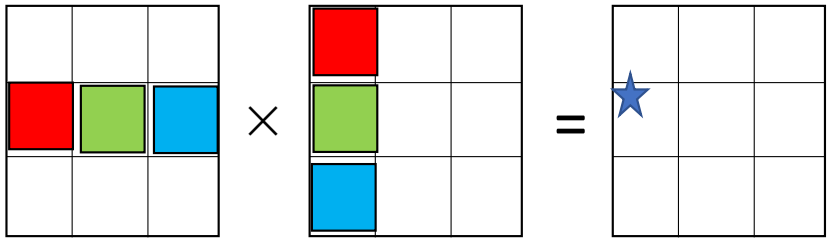
# Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {
        if(/* in bounds */)
                Mds[threadIdx.y][threadIdx.x] = …;
        if(/* in bounds */)
                Nds[threadIdx.y][threadIdx.x] = …;
         __syncthreads();
}
```

Step 3a: Load cell of cached submatrices (then wait)

# Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {
        if(/* in bounds */)
                Mds[threadIdx.y][threadIdx.x] = …;
        if(/* in bounds */)
                Nds[threadIdx.y][threadIdx.x] = …;
         __syncthreads();

        for(k=0; k < TILE_WIDTH; k++)
                tmp += …
         __syncthreads();
}
```

Step 3a: Load cell of cached submatrices (then wait)

Step 3b: Calculate submatrix contribution (then wait)

# Writing the tiled kernel

```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        …
         for (int m=0; m < num_tiles; m++) {
                …
        }

        if (row < width && col < width)
                Pd[row*width+col] = tmp;
}
```

Step 4: Write the value into the appropriate cell

# Mapping modules to a CS curriculum

[A1] Heterogeneous Computing: Elementary Notions

[A2] Task Mapping on Soft Heterogeneous Systems

[B1] Hybrid Algorithms

[A3] Pollack's Rule

[C1] Introduction to ARM

[D1] Introduction to CUDA Programming

[C2] GPU Memory Hierarchy

[D2] Heterogeneous Programming with OpenMP