Programming Parallel Computers 2019

Jukka Suomela · Jaakko Lehtinen · Samuli Laine

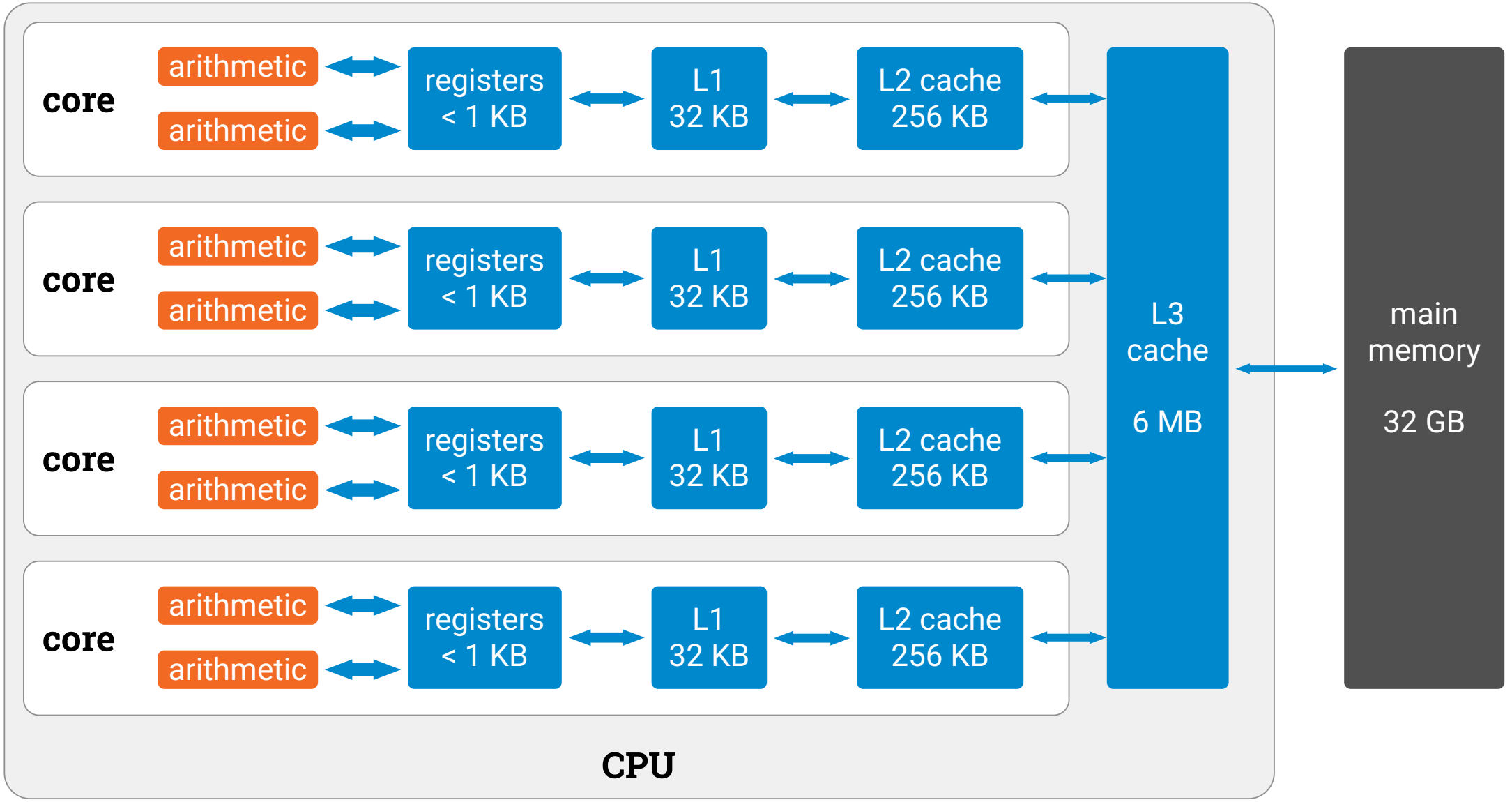
Aalto University

ppc.cs.aalto.fi

week 4

Quick recap

Programming modern CPUs

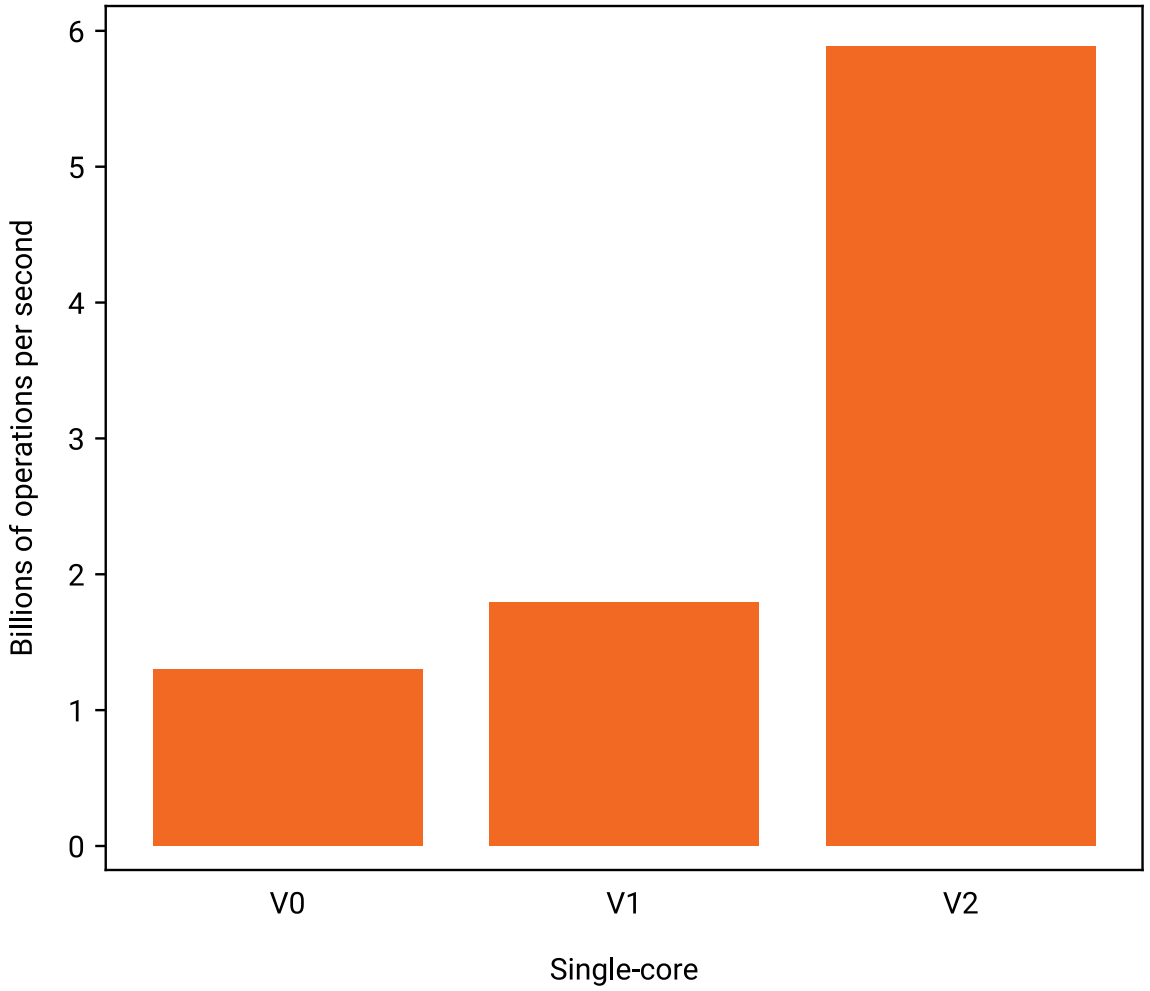

core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
L3
cache
6 MB
main
memory
32 GB
CPU

ILP

V0: baseline

V1: linear reading

V2: instruction-level parallelism

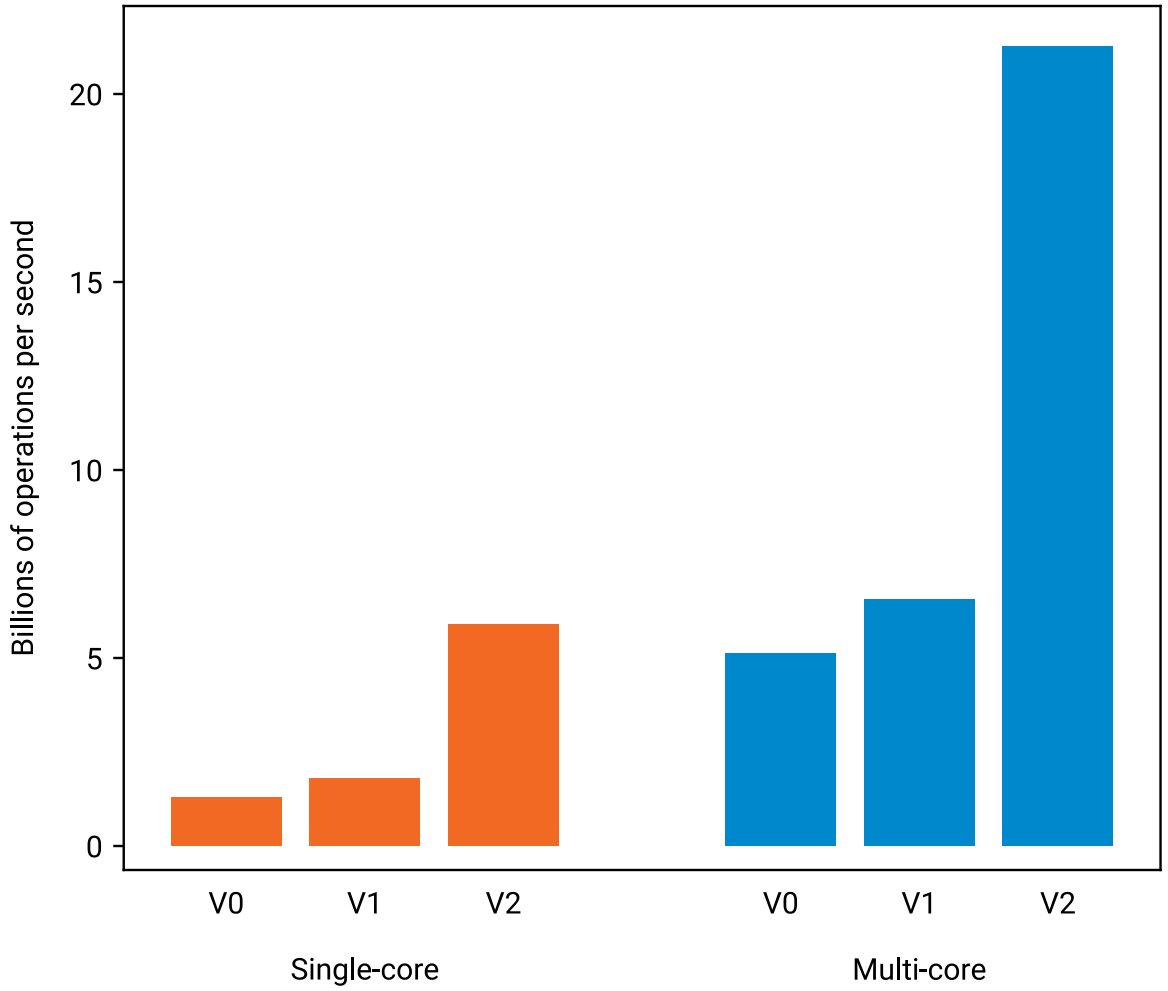


Multi-core parallelism

V0: baseline

V1: linear reading

V2: instruction-level parallelism



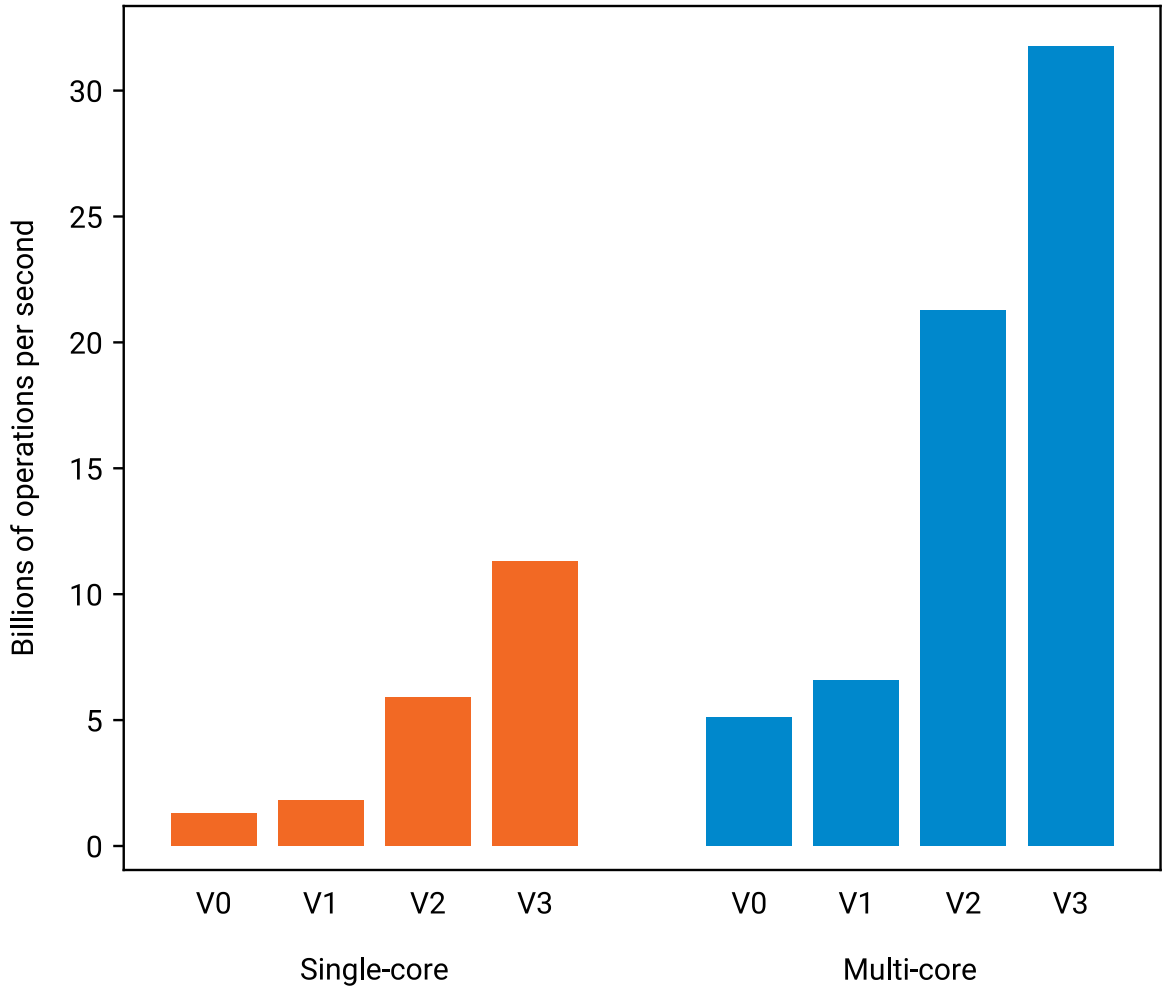
Vector operations

V0: baseline

V1: linear reading

V2: instruction-level parallelism

V3: vectorization



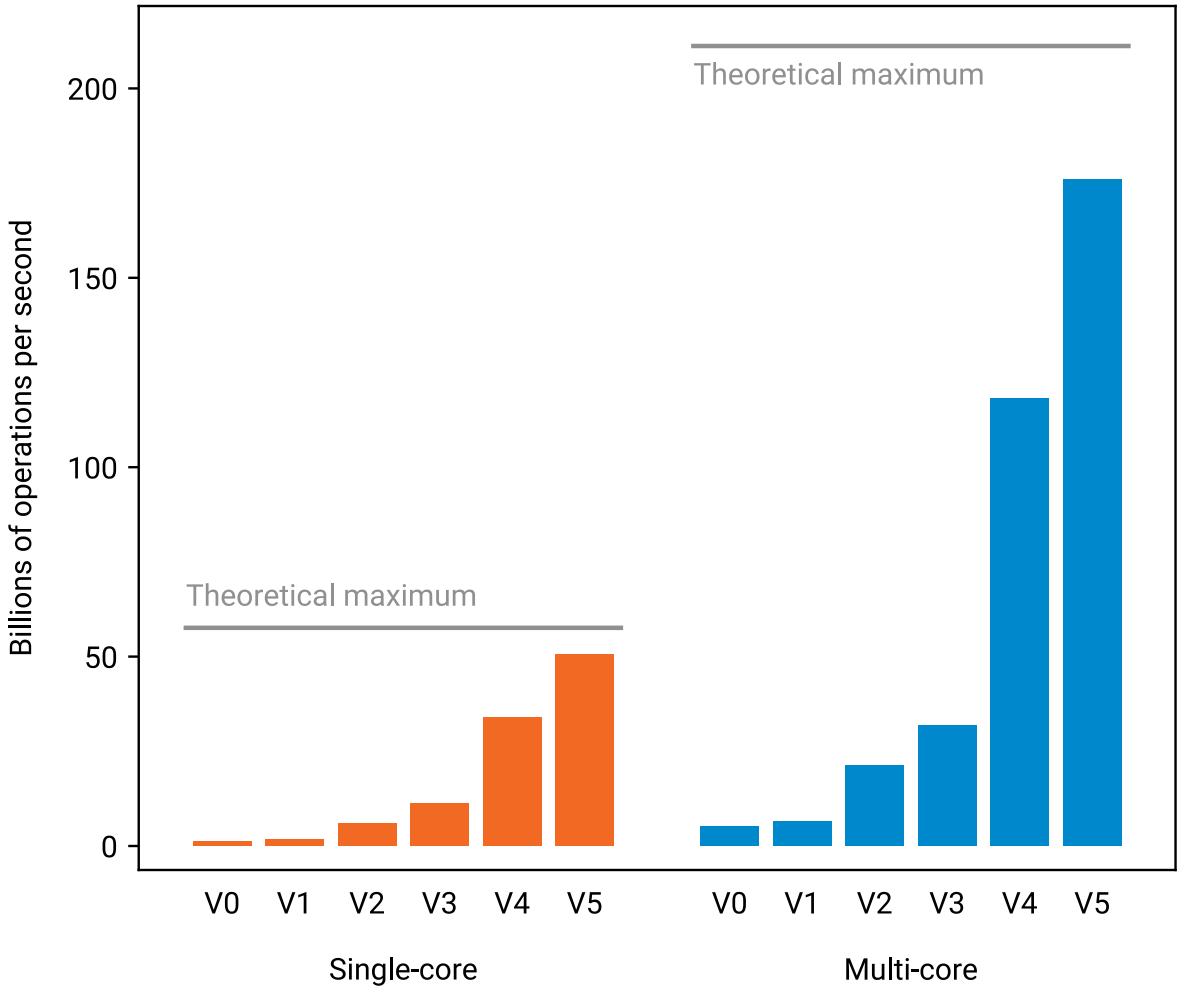
Reuse data in registers

...

V3: vectorization

V4: reuse data

V5: more data reuse



Reuse data in caches

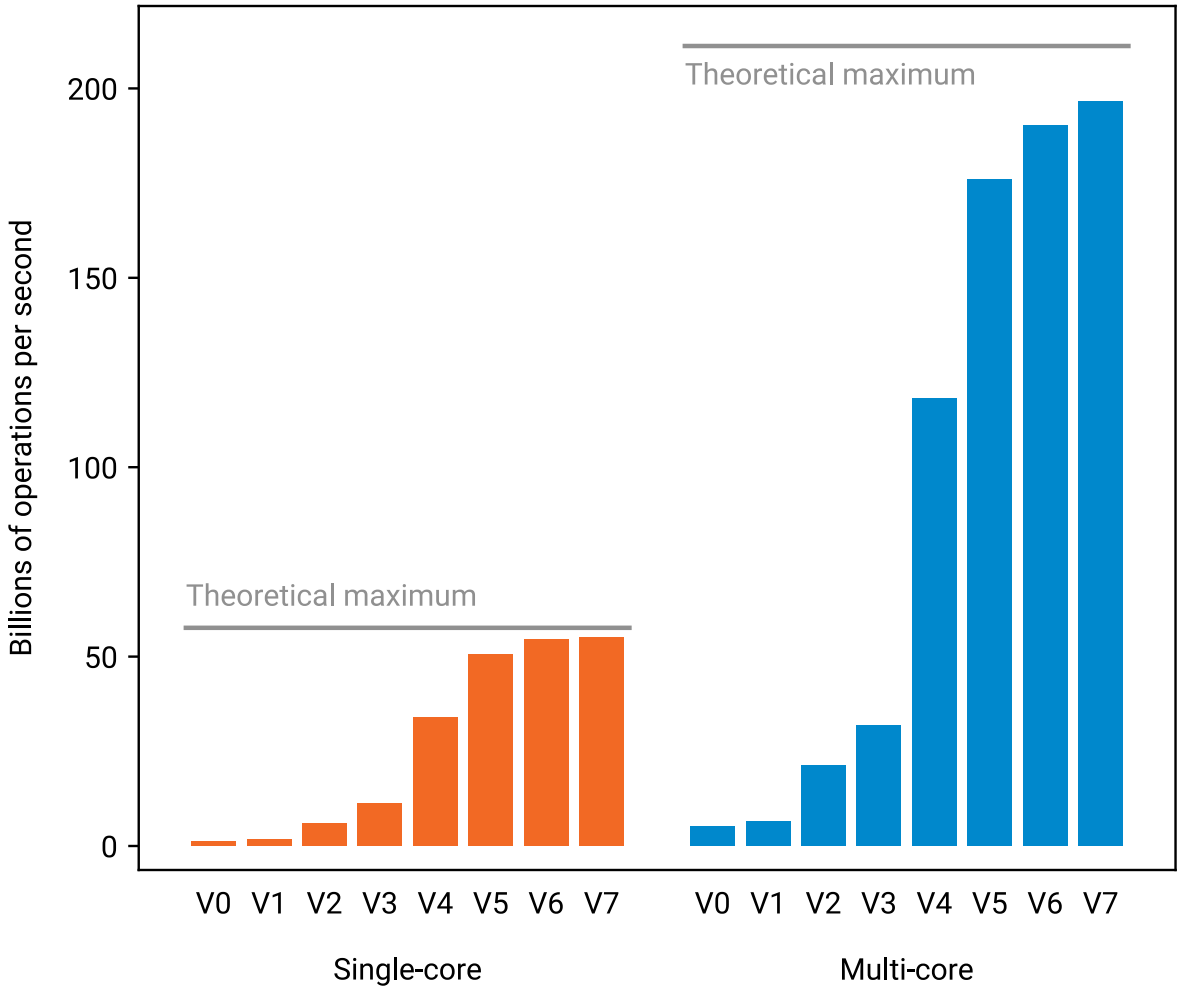
...

V4: reuse data

V5: more data reuse

V6: prefetching

V7: cache blocking



Some key ideas

- Design algorithms so that there are lots of ***independent operations***
  - needed for **any kind of parallelism**
- Preferably lots of ***similar*** independent operations
  - needed for **vector parallelism**
- Try to do ***lots of arithmetic operations per memory access***
  - otherwise the CPU will be mostly idle, **waiting for some data to process**

GPU programming

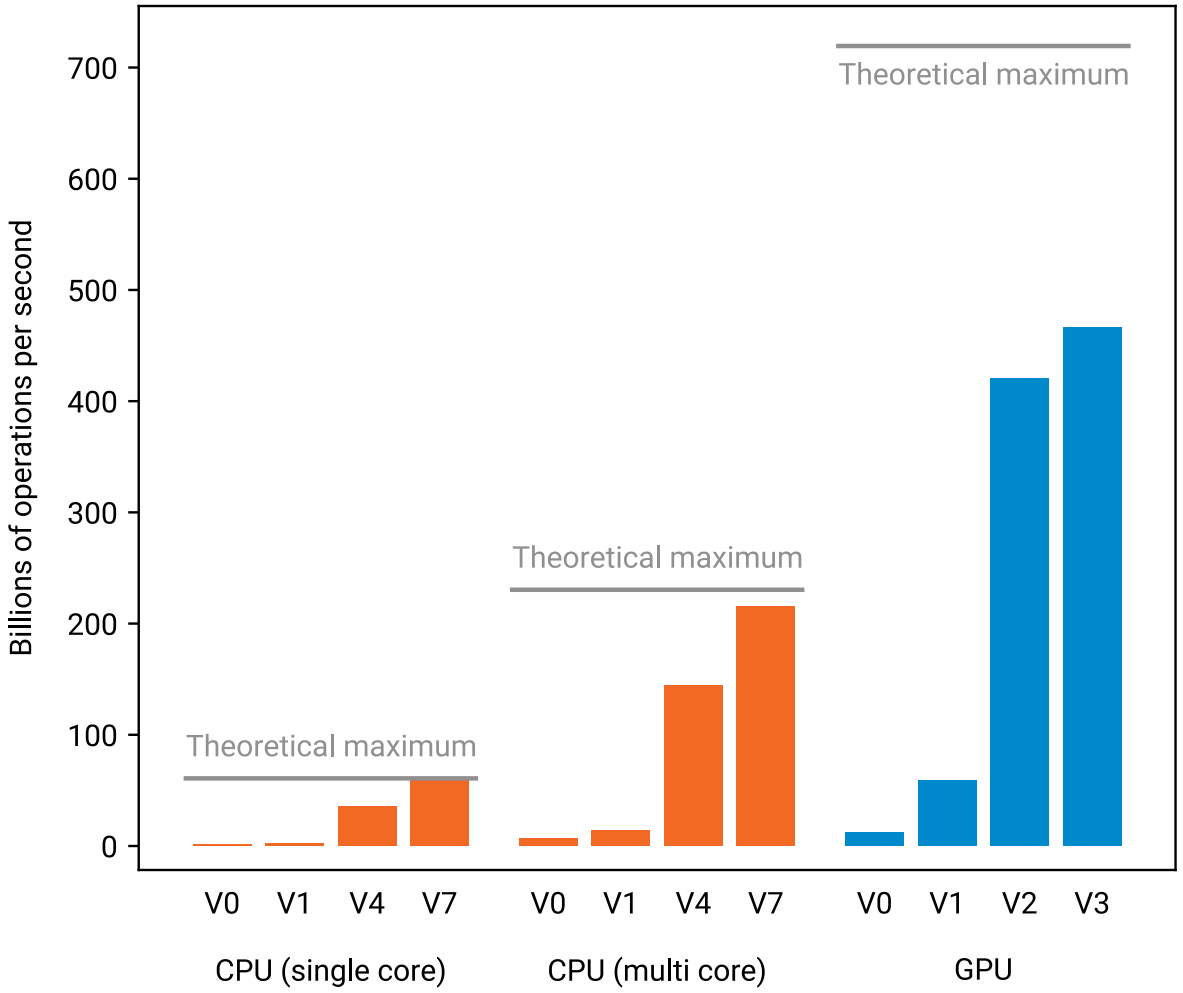
GPU — graphics processing unit

- All modern computers have at least two processors:
  - ***CPU***: what we have been using so far
  - ***GPU***: lots of more computing power available and we can use it, too!
- CPU on Maari computers :
  - **460 billion** single-precision FLOPS (floating-point operations / second)
  - **230 billion** double-precision FLOPS
- GPU on Maari computers:
  - **1400 billion** single-precision FLOPS
  - **45 billion** double-precision FLOPS

GPU computing

We will continue with the same sample problem

We will develop GPU solutions that outperform our best CPU solutions



Parallel computing resources

CPU

- Pipelined arithmetic units:
  1 new operation per cycle
- ***Vector*** operations:
  8 similar operations
- Lots of arithmetic units:
  4 × 2 vector operations in parallel e.g. for FMA

GPU

- Pipelined arithmetic units:
  1 new operation per cycle
- "***Warp***" of "***threads***":
  32 similar operations
- Lots of arithmetic units:
  5 × 4 warps executed in parallel e.g. for FMA

Parallel computing resources

CPU

- 64 single-precision arithmetic units
- 32 double-precision arithmetic units
- 3.4–3.8 GHz

GPU

- 640 single-precision arithmetic units
- 20 double-precision arithmetic units
- 1.0–1.1 GHz

Differences in programming models

CPU

- **“SIMD”** — single instruction, multiple data
- **One thread**, *vector registers* x, y, z
- **One thread says:** please calculate `z[i] = x[i] + y[i]`

GPU

- **“SIMT”** — single instruction, multiple thread
- **Many threads**, each with *scalar registers* x, y, z
- **All threads say:** please calculate `z = x + y`

Differences in programming models

CPU

- **“SIMD”** — single instruction, multiple data
- Memory access:
  read full vector of data
  - preferably properly aligned

GPU

- **“SIMT”** — single instruction, multiple thread
- Memory access:
  each thread reads one scalar
  - preferably inside a small number of cache lines

Differences in programming models

CPU

- **“SIMD”** — single instruction, multiple data
- Some SIMT solutions are hard to implement here

GPU

- **“SIMT”** — single instruction, multiple thread
- Any SIMD solution easy to implement here

How to keep pipelines full

CPU

- Instructions: **4–8** threads
- Resources: **8** vector operations
- CPU ***looks at the instruction stream far into the future***, finds instructions that are ready for execution

GPU

- Instructions: **huge number** of warps available for execution
- Resources: **20** warp operations
- GPU only looks at the ***first instruction of each warp***

How to keep pipelines full

CPU

- Tries to run old sequential code reasonably well
- Lots of complicated hardware to support that
  - out-of-order execution
  - high clock frequency
  - many layers of cache

GPU

- Does not care anything about running old sequential code
- Simpler hardware
  - transistors used for arithmetic, not for control logic
  - easier to add more parallel units than increase clock speed

Programmer's view

CPU

- ***Instruction-level parallelism important***
- Everything else is sequential unless explicitly parallelized

```
#pragma omp
float8_t
```

GPU

- Instruction-level parallelism not so important
- The ***only*** primitive that we can use is inherently parallel

```
f<<<blocks, threads>>>()
```

GPU programming

- We will use NVIDIA GPUs and ***CUDA*** programming environment
  - CUDA is basically an extension of C++
  - you can write normal C++ code
  - unless said otherwise, your code runs on **CPU**
  - whenever needed, you can say "***please launch lots of threads that run this function on GPU in a massively parallel fashion***"
- Other tools exists
  - cross-platform tool: ***OpenCL***
  - can run the same code on e.g. NVIDIA and AMD GPUs
  - see course material for examples

CUDA programming model

Parallel GPU solution

```
__global__ void mykernel() {
    int i = blockIdx.x;
    int j = threadIdx.x;
    foo(i, j);
}

int main() {
    mykernel<<<100, 128>>>();
}
```

Equivalent sequential code

```
for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < 128; ++j) {
        foo(i, j);
    }
}
```

Launch 100 × 128 threads on GPU

Hardware overview

Example

Our first GPU program

Toy example: split evenly

- What is the best way to **split $1^5, 2^5, 3^5, \ldots, 30^5$ in two parts** such that their **sums** are as close to each other as possible?
- We will solve this with a **naive brute force algorithm**
  - first with CPUs
  - then with GPUs
- Algorithms: just try out all **2^{30} cases** and see what is best
  - each case is represented as a **30-bit binary number**
  - bit **0** in position $x - 1$: number x^5 in the **first part**
  - bit **1** in position $x - 1$: number x^5 in the **second part**

```
inline int p5(int i) { return i * i * i * i * i; }

inline int value(int x) {
    int a = 0;
    for (int i = 0; i < 30; ++i) {
        if (x & (1 << i)) {
            a += p5(i+1);
        } else {
            a -= p5(i+1);
        }
    }
    return abs(a);
}
```

Find $0 \le x < 2^{30}$ that minimizes value(x)

Sequential CPU solution

```
constexpr int total = 1 << 30;
int best_x = 0;
int best_v = value(best_x);
for (int x = 0; x < total; ++x) {
    int v = value(x);
    if (v < best_v) {
        best_x = x;
        best_v = v;
    }
}
```

Find $0 \le x < 2^{30}$ that minimizes value(x)

GPU: splitting work

- We will need to have lots of "***blocks***" ready for execution
- Each block has to have lots of "***threads***"
  - GPU will split blocks in "**warps**" that consist of 32 threads
  - reasonable block sizes are small multiples of 32
  - e.g. 64, 128, 256
- Each thread does some reasonable amount of work

Our choices:

- **2^{10} = 1024 blocks**
- **2^{6} = 64 threads per block**
- **2^{14} = 16K iterations per thread**

GPU: splitting work

- **One unit of work** = one 30-bit number
- ***Block index*** = highest 10 bits
- ***Thread index*** = next 6 bits
- **Iteration** = lowest 14 bits

Our choices:

- 2^{10} = 1024 blocks
- 2^{6} = 64 threads per block
- 2^{14} = 16K iterations per thread

```
__global__ void mykernel(int* r) {
    int x3 = blockIdx.x;
    int x2 = threadIdx.x;
    int best_x = 0;
    int best_v = value(best_x);
    for (int x1 = 0; x1 < iterations; ++x1) {
        int x = (x3 << 20) | (x2 << 14) | x1;
        int v = value(x);
        if (v < best_v) {
            best_x = x;
            best_v = v;
        }
    }
    r[(x3 << 6) | x2] = best_x;
}
```

What is my part of search space?

Mostly normal sequential C++ code

Save best solution in my part of search space

```
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int),
           cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that
// minimizes value(x)
```

Allocate GPU memory for result

Launch 2^{16} threads on GPU

Copy result back from GPU memory to CPU memory

```
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int),
           cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that
// minimizes value(x)
```

Error checking omitted!

Try it out

- Compile & link with “**nvcc**” instead of “**g++**”
- Run as usual
  - sequential CPU solution: **38 seconds**
  - parallel GPU solution: **0.3 seconds**

$$1^5 + 2^5 + 3^5 + 4^5 + 5^5 + 9^5 + 10^5 + 12^5 + 15^5 + 16^5 + 17^5 + 19^5 + 22^5 + 23^5 + 24^5 + 25^5 + 27^5 + 28^5 = 66993712$$

$$6^5 + 7^5 + 8^5 + 11^5 + 13^5 + 14^5 + 18^5 + 20^5 + 21^5 + 26^5 + 29^5 + 30^5 = 66993713$$

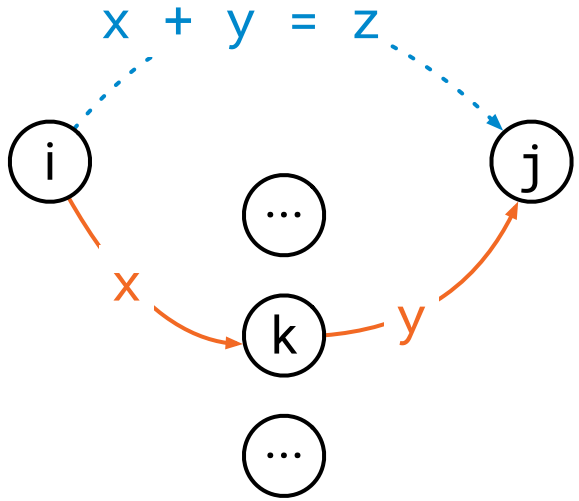
Typical program structure

- ***GPU side:***
  - “**kernel**” that does one small part of work
- ***CPU side:***
  - allocate GPU memory for input & output
  - copy input from CPU memory to GPU memory
  - **launch kernel** (lots of blocks, lots of threads)
  - copy result back from GPU memory to CPU memory
  - release GPU memory

Case study

Cheapest 2-hop path

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```



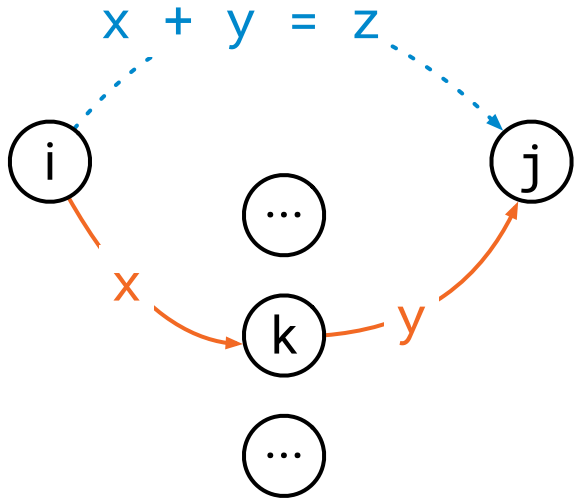
Splitting work

- One thread: innermost loop
  - n units of work
  - calculates 1 result
- One block: 16 × 16 threads
  - 16 × 16 × n units of work
  - calculates 16 × 16 results
- Number of blocks: (n/16) × (n/16), rounded up

```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

What if n is not a multiple of 16

blockDim.x = 16
blockDim.y = 16



```
float* dGPU = NULL;
cudaMalloc((void**)&dGPU, n * n * sizeof(float));
float* rGPU = NULL;
cudaMalloc((void**)&rGPU, n * n * sizeof(float));
cudaMemcpy(dGPU, d, n * n * sizeof(float),
           cudaMemcpyHostToDevice);

dim3 dimBlock(16, 16);
dim3 dimGrid(divup(n, dimBlock.x), divup(n, dimBlock.y));
mykernel<<<dimGrid, dimBlock>>>(rGPU, dGPU, n);

cudaMemcpy(r, rGPU, n * n * sizeof(float),
           cudaMemcpyDeviceToHost);
cudaFree(dGPU); cudaFree(rGPU);
```

Performance

- Test input: ***n* = 6300**
- Maari computers:
  - baseline CPU solution: **397 s**
  - best CPU solution: **2.3 s**
  - current GPU solution: **42 s**
- What is the bottleneck?

Memory access pattern

- Blocks are divided in warps
  - warp = 32 threads
- ***Entire warp executes synchronously***
- If one thread reads some memory,
  all threads of the warp read some memory
  - threads access small continuous parts of memory:
    need to load few cache lines → **good**
  - threads access 32 different locations far from each other:
    need to load many cache lines → **bad**

First warp:

thread 0: i = 0, j = 0

thread 1: i = 1, j = 0

thread 2: i = 2, j = 0

thread 3: i = 3, j = 0

…

thread 31: i = 15, j = 1

Pay attention to this index!

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

Good

First warp, first iteration:

thread 0: read d[0]
thread 1: read d[1000]
thread 2: read d[2000]
thread 3: read d[3000]
…

thread 0: read d[0]
thread 1: read d[0]
thread 2: read d[0]
thread 3: read d[0]
…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

Good

First warp, second iteration:

thread 0: read d[1]
thread 1: read d[1001]
thread 2: read d[2001]
thread 3: read d[3001]
…

thread 0: read d[1000]
thread 1: read d[1000]
thread 2: read d[1000]
thread 3: read d[1000]
…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

Bad

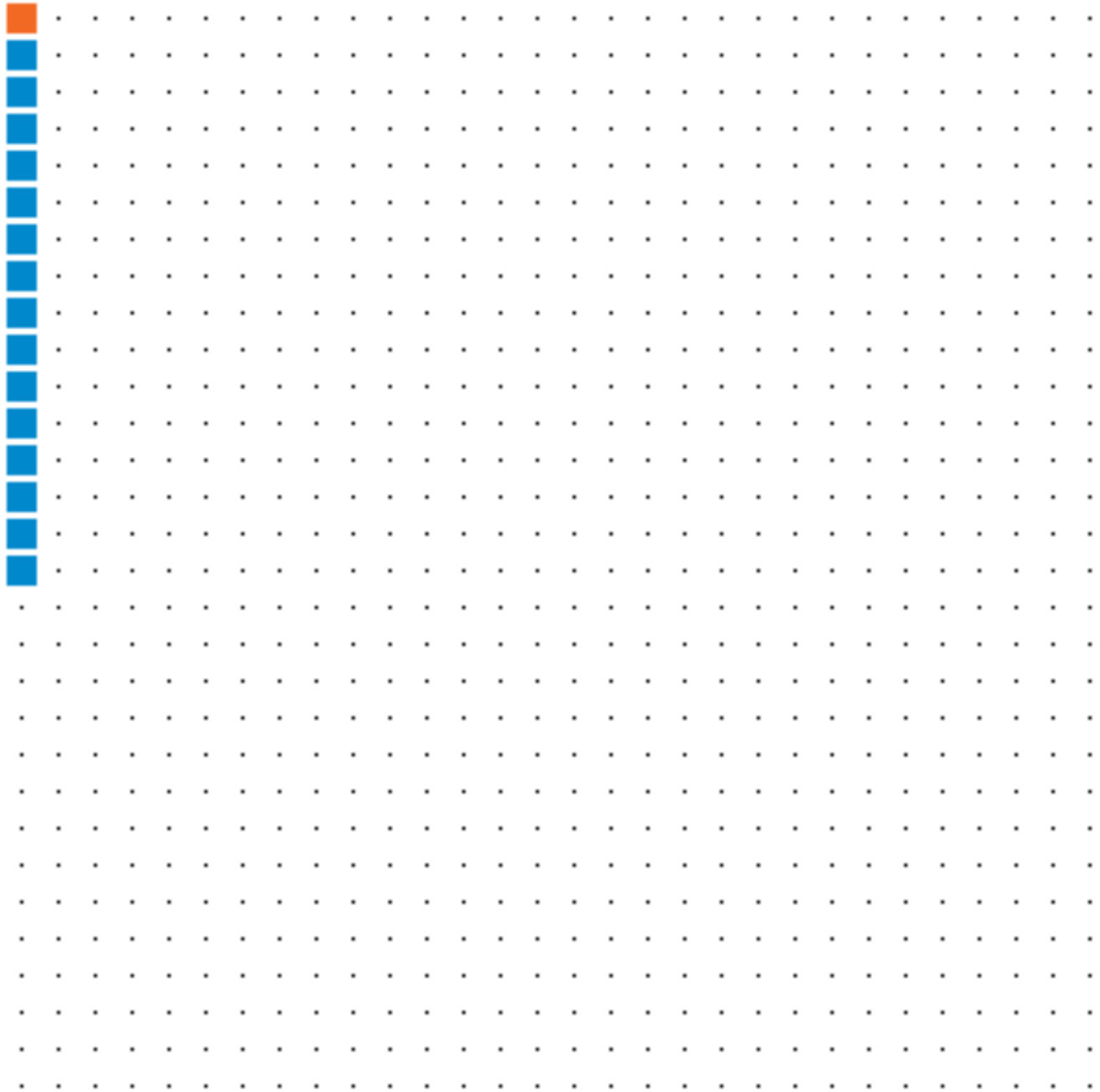
Good

First warp, third iteration:

thread 0: read d[2]
thread 1: read d[1002]
thread 2: read d[2002]
thread 3: read d[3002]
…

thread 0: read d[2000]
thread 1: read d[2000]
thread 2: read d[2000]
thread 3: read d[2000]
…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```



```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    ...
}
```

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Good

First warp, first iteration:

thread 0: read d[0]
thread 1: read d[0]
thread 2: read d[0]
thread 3: read d[0]
…

Good

thread 0: read d[0]
thread 1: read d[1]
thread 2: read d[2]
thread 3: read d[3]
…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Good

Good

First warp, second iteration:

thread 0: read d[1]

thread 1: read d[1]

thread 2: read d[1]

thread 3: read d[1]

…

thread 0: read d[1000]

thread 1: read d[1001]

thread 2: read d[1002]

thread 3: read d[1003]

…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Good

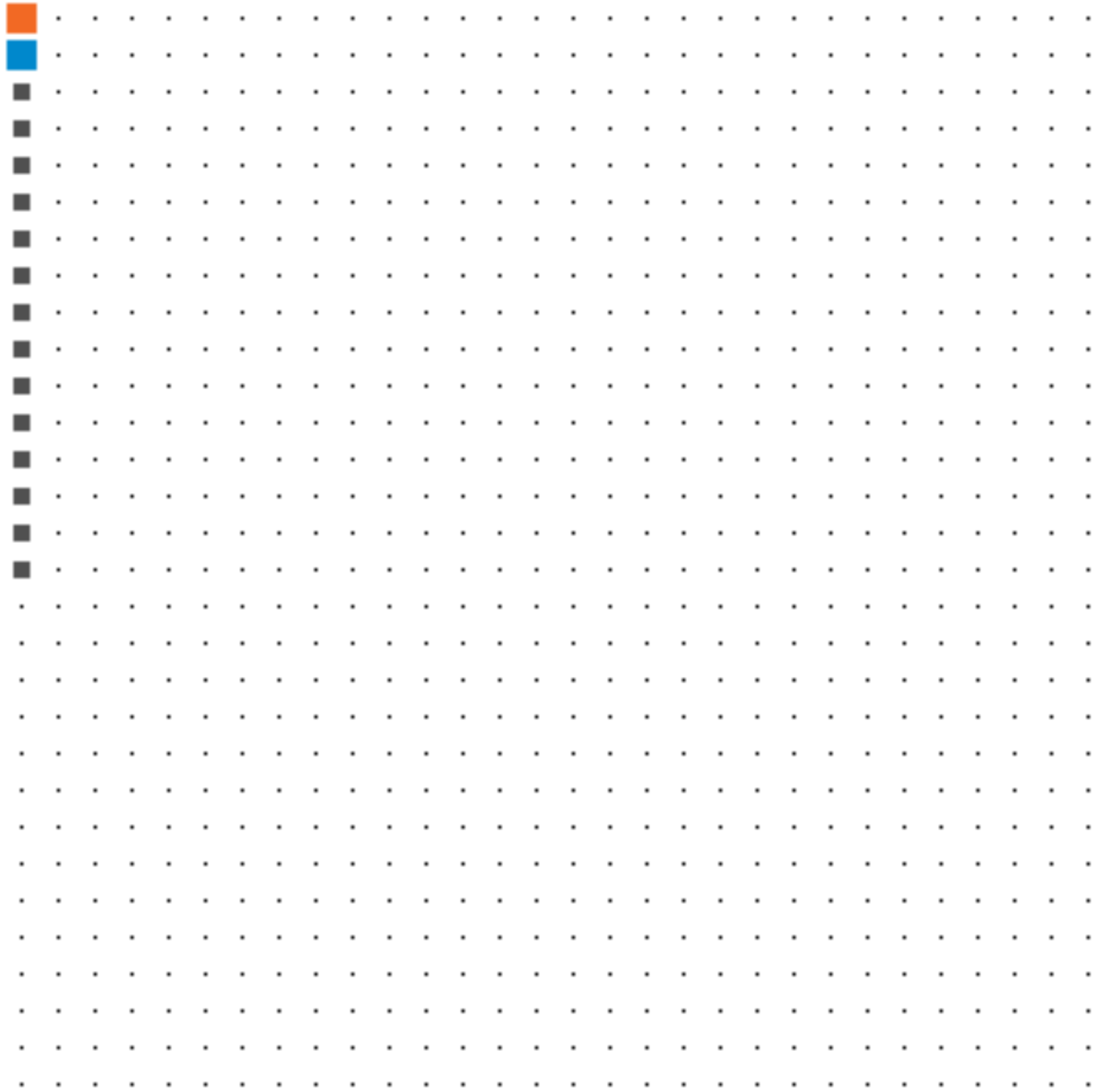
Good

First warp, second iteration:

thread 0: read **d[2]**
thread 1: read **d[2]**
thread 2: read **d[2]**
thread 3: read **d[2]**
…

thread 0: read **d[2000]**
thread 1: read **d[2001]**
thread 2: read **d[2002]**
thread 3: read **d[2003]**
…

```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```



```
int i = threadIdx.x + ...
int j = threadIdx.y + ...
for (... ++k) {
    float x = d[n*i + k];
    float y = d[n*k + j];
    float x = d[n*j + k];
    float y = d[n*k + i];
    ...
}
```

Performance

- **V0:** baseline — **42 s**
- **V1:** better memory access pattern — **8 s**
- But we can do much better by applying familiar ideas:
  - **reuse data in registers**
  - **reuse data in “cache”** (here: shared memory)

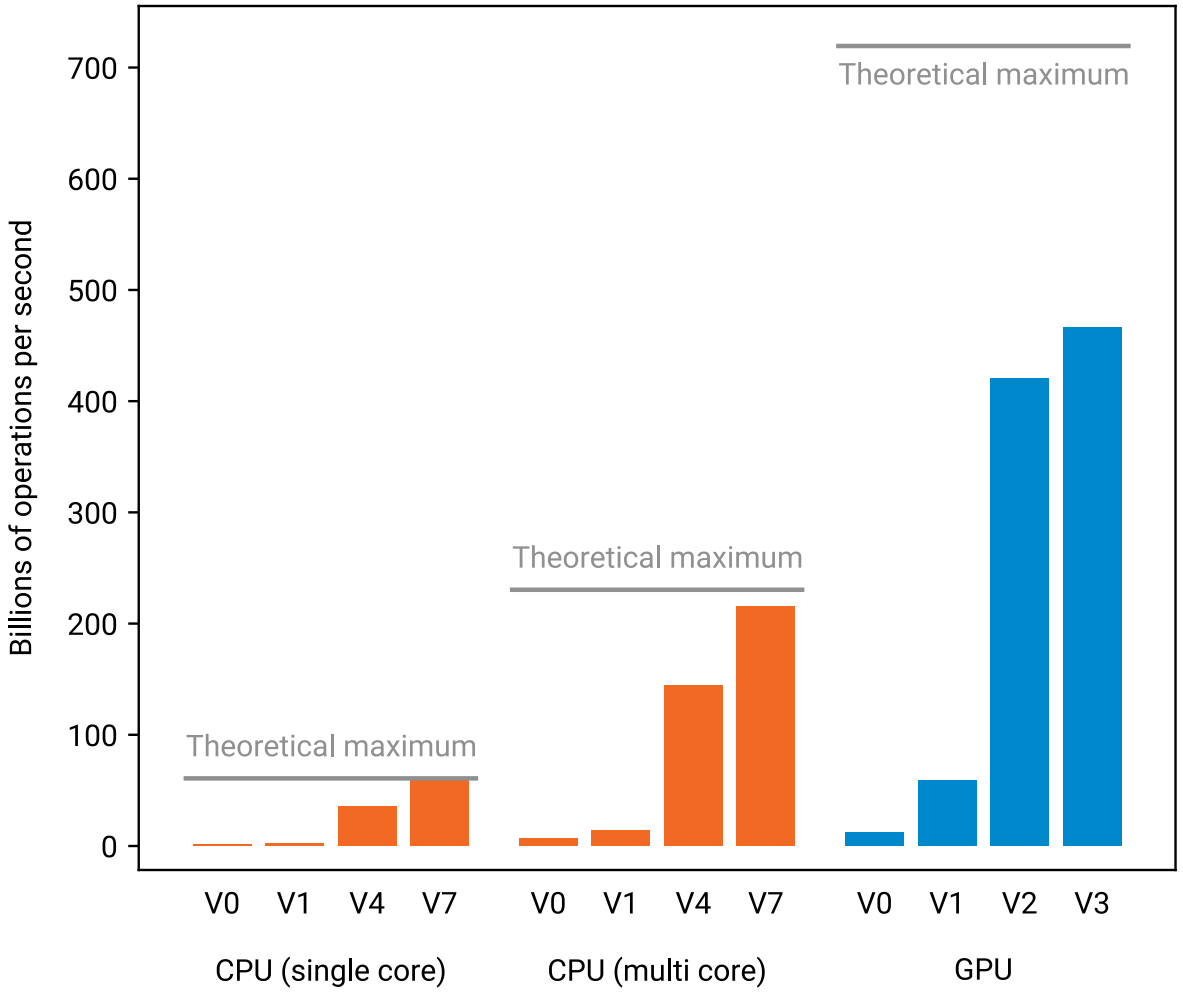
Performance

V0: baseline

V1: memory access

V2: registers

V3: shared memory



Coming next week

- ***Samuli Laine (NVIDIA):***
  advanced GPU programming