CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 25: Data-level Parallelism

Instructors: Dan Garcia, Justin Yokota

Agenda

- Why Parallelism?
- Data Level Parallelism: SIMD
- Applying SIMD to Matrix Multiplication

Computing in the News

Cleaning Up the Atmosphere with Quantum Computing

The amount of carbon dioxide in the atmosphere increases daily with no sign of stopping or slowing. Too much of civilization depends on the burning of fossil fuels, and even if we can develop a replacement energy source, much of the damage has already been done. Without removal, the carbon dioxide already in the atmosphere will continue to wreak havoc for centuries.

Atmospheric carbon capture is a potential remedy to this problem. It would pull carbon dioxide out of the air and store it permanently to reverse the effects of climate change. Practical carbon capture technologies are still in the early stages of development, with the most promising involving a class of compounds called amines that can chemically bind with carbon dioxide. Efficiency is paramount in these designs, and identifying even slightly better compounds could lead to the capture of billions of tons of additional carbon dioxide.

In AVS Quantum Science, by AIP Publishing, researchers from the National Energy Technology Laboratory and the University of Kentucky deployed an algorithm to study amine reactions through quantum computing. The algorithm can be run on an existing quantum computer to find useful amine compounds for carbon capture more quickly.

“We are not satisfied with the current amine molecules that we use for this [carbon capture] process,” said author Qing Shao. “We can try to find a new molecule to do it, but if we want to test it using classical computing resources, it will be a very expensive calculation. Our hope is to have a fast algorithm that can screen thousands of new molecules and structures.”

Any computer algorithm that simulates a chemical reaction needs to account for the interactions between every pair of atoms involved. Even a simple three-atom molecule like carbon dioxide bonding with the simplest amine, ammonia, which has four atoms, results in hundreds of atomic interactions. This problem vexes traditional computers but is exactly the sort of question at which quantum computers excel.

Why Parallelism?

- To answer this question, it's useful to take a look at modern supercomputers
- Often large warehouse-scale systems, which when used properly can handle extremely fast/large computations
  - Berkeley's Savio Cluster
  - Google/Amazon computing warehouses
- Personal computers will generally have similar structures at a smaller scale
- My computer:
  - 2 GHz clock cycle
  - 16 GB RAM
  - 64-bit Intel CPU using x64 (the 64-bit version of x86 assembly)

What is the biggest difference between a supercomputer and my computer?

- A supercomputer has a higher clock speed
- A supercomputer has faster memory accesses
- A supercomputer has a higher transistor density
- A supercomputer can perform more complicated operations in a single clock cycle
- Nothing; a supercomputer is just a bunch of regular computers wired together

What is the biggest difference between a supercomputer and my computer?

- A supercomputer has a higher clock speed
- A supercomputer has faster memory accesses
- A supercomputer has a higher transistor density
- A supercomputer can perform more complicated operations in a single clock cycle
- Nothing; a supercomputer is just a bunch of regular computers wired together

What is the biggest difference between a supercomputer and my computer?

- A supercomputer has a higher clock speed
- A supercomputer has faster memory accesses
- A supercomputer has a higher transistor density
- A supercomputer can perform more complicated operations in a single clock cycle
- Nothing; a supercomputer is just a bunch of regular computers wired together

Higher Clock Speed?

- The speed of light is ~300,000,000 m/s
  - 1 GHz = 1/1,000,000,000 s, so 1 clock cycles per nanosecond
  - 3e8 m/s * 1s/1e9 ns = 0.3 m/ns
- Light travels 30 cm = about one foot in one nanosecond
- My clock cycle is 2 GHz, which is a clock cycle every half-nanosecond
- Conclusion: My computer's running so fast that the concept of simultaneity would break down if my CPU was larger than my hand
- A warehouse is larger than my hand (I hope), so it would be physically impossible for a supercomputer to run at a significantly faster clock speed

Faster Memory Accesses/More Transistors?

- Not significantly different
- Magnetic-disk based memory accesses also are near physical limits
- More transistors = more heat = silicon starts to melt
- Small improvements possible, but not ~1000x better.

More complicated operations?

- Depends on the architecture
  - For RISC-V, it's counterproductive to have more complicated operations
  - For CISC architectures, this is actually feasible
- This leads to SIMD operations (today's topic)
- Still can only get ~4-8x better results than my PC, so relatively low effect.

More computers?

- This is the biggest difference.
- My computer has 12 independent CPUs that can run separate programs
- The Savio cluster has 3600 CPUs, with each CPU just as powerful as one of my CPUs
- Therefore, in order to gain any benefits from using a supercomputer, we need to know how to get many computers to work together on the same problem

Why Parallelism?

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Time}}{\text{Cycle}}$$

- Overall, our goal is to continue increasing the amount of computation that can be done per unit time.
- Recall the "Iron Law" of Processor Performance, which dictates the speed a program runs on one processor. In order to speed up our code, we need to improve one of:
- Instructions/Program
  - Either we reduce the work we do to solve the problem, or
  - We increase the amount of work we do per instruction
- Cycles/Instruction
- Time/Cycle

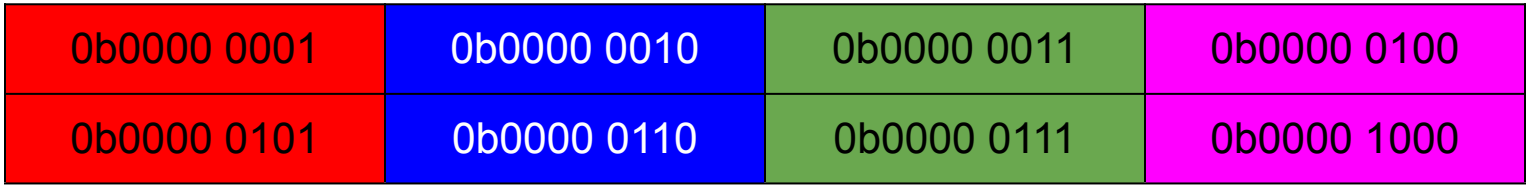
Toy Example: Vector Sum

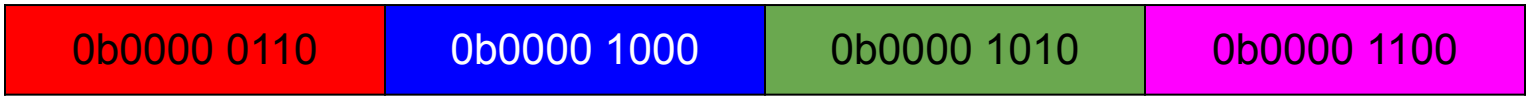
0b0000 0001	0b0000 0010	0b0000 0011	0b0000 0100
0b0000 0101	0b0000 0110	0b0000 0111	0b0000 1000

0b0000 0110	0b0000 1000	0b0000 1010	0b0000 1100

- We have two 4-D vectors whose components are 8-bit numbers
- Goal: Determine the sum of the two vectors
- For the following example, inputs stored at 0(a0) and 0(a1), and output saved to 0(a2)

Toy Example: Vector Sum: Naive






```
lb t0 0(a0)
lb t1 0(a1)
add t0 t0 t1
sb t0 0(a2)
```

```
lb t0 1(a0)
lb t1 1(a1)
add t0 t0 t1
sb t0 1(a2)
```

```
lb t0 2(a0)
lb t1 2(a1)
add t0 t0 t1
sb t0 2(a2)
```

```
lb t0 3(a0)
lb t1 3(a1)
add t0 t0 t1
sb t0 3(a2)
```

- 16 total instructions. Can we do better?

Toy Example: Vector Sum: Single Add

0b0000 0001	0b0000 0010	0b0000 0011	0b0000 0100
0b0000 0101	0b0000 0110	0b0000 0111	0b0000 1000

0b0000 0110	0b0000 1000	0b0000 1010	0b0000 1100

- Solution: If we treat these arrays as 32-bit integers, we can add with one operation, and do this in 4 instructions.

```
lw t0 0(a0)
lw t1 0(a1)
add t0 t0 t1
sw t0 0(a2)
```

Toy Example: Vector Sum: Vectorized Add

0b0000 0001	0b0000 0010	0b0000 0011	0b0000 0100
0b0000 0101	0b0000 0110	0b0000 0111	0b0000 1000

0b0000 0110	0b0000 1000	0b0000 1010	0b0000 1100

- This doesn't quite work, because overflow on one element affects other elements. So we need to create a slightly different instruction that ignores overflow every 8th bit.
- New instruction should take about as long as a single add instruction, since the circuit's similar

```
lw t0 0(a0)
lw t1 0(a1)
vec_add t0 t0 t1
sw t0 0(a2)
```

SIMD Instructions

- Instead of doing math on one number at a time, we can instead do math on several numbers at a time, in a single clock cycle
- Known as SIMD instructions (Single-Instruction, Multiple Data) or vector instructions
- Use specialized "vector" registers which store 128, 256, or even 512 bits
- SIMD instructions act as extensions to the base instruction set, with different systems supporting different SIMD instructions.
- Generally speaking, most of the speedup comes not from doing four math operations at a time, but instead from doing a large memory load/store at a time.

SIMD Instructions

- Caveats: Each instruction needs its own circuitry, so we're limited to the set of instructions that came with the CPU
  - RISC-V doesn't have a standard vector library, so we're using x86's vector operators here.
  - In practice, this doesn't matter too much since arithmetic syntax works similarly to RISC-V
- There's still only one PC, so we can't vectorize branch or jump instructions
  - Lab discusses a way to get around that using the equivalent of slt
- Since we only have limited instructions available, we can't do different math operations to vector components, and we can only easily load consecutive blocks of memory to a vector
  - Note that most programs spend the majority of their time loading/storing instead of doing math, because loads and stores can take hundreds of cycles to resolve
- Large vectors require significant amounts of circuitry, so they are expensive to implement, and often have higher cycles/instruction than standard instructions

Intel Intrinsics

- SSE library
  - 64- and 128-bit registers: 4 32-bit integers at a time or 2 doubles at a time
- AVX library
  - 256-bit registers: 8 32-bit integers at a time or 4 doubles at a time
- AVX-2 library
  - Extension of the AVX library, with more supported instructions
- AVX-512 library
  - 512 bit registers
  - Big enough that it tends to cause throttling issues, so Intel's been quietly trying to kill this library
  - Not covered in this class, and not available on hive machines.

Intel Intrinsics: Types

- `__m256`
  - 256 bit register for storing floats
- `__m256d`
  - 256 bit register for storing doubles
- `__m256i`
  - 256 bit register for storing 32-bit integers
- `__m128, __m128d, __m128i`
  - 128 bit registers
- Each type corresponds directly to a type of SIMD register (note that x86 has different sets of registers for floats, doubles, and integers).
- Used similarly to variables, but directly are associated with available registers, so you can't just initialize a bunch of them (or an array of them)

Intel Intrinsics: Instructions

- Generally of the format:
- `_<register size>_<instruction>_<component_type>`
- Ex. `_mm256_add_epi32` adds two 256-bit vectors, treating the vectors as arrays of 32-bit integers.
- Ex. `_mm_load_ps` loads 4 consecutive floats into a 128-bit register from the given memory address. The memory address must be aligned to a 16-byte boundary (loadu allows for nonaligned addresses, but is slower)
- More instructions: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- Looks like C functions, but programming with them feels more like assembly

Vector Sum (128-bit registers, 32-bit integers

0x0000 0001	0x0000 0002	0x0000 0003	0x0000 0004
0x0000 0005	0x0000 0006	0x0000 0007	0x0000 0008

0x0000 0006	0x0000 0008	0x0000 000A	0x0000 000C

```
__mm128i avec = _mm_load_si128(a);
__mm128i bvec = _mm_load_si128(b);
__mm128i sum = _mm_add_epi32(avec, bvec);
_mm_store_si128(c, sum);
```

Common mistakes when working with SIMD instructions

- Trying to directly access a 32-bit chunk of a SIMD vector (such as through typecasting)
  - Need to do an explicit load/store, since registers are different from memory
- Trying to _mm_load or _mm_store with unaligned addresses
  - Use loadu or storeu if you must, or try to get your addresses aligned
  - For mallocs, aligned_alloc gives you an aligned address
  - For local variables, you can set an attribute (example shown in slides)
- Forgetting the tail case
  - If your data isn't an array whose length is a multiple of your vector size, you need to handle the last iterations of your dataset one-by-one instead of 4 at a time.
- Using too many vectors (or creating a large array of vectors)
  - Ends up throttling your code because the compiler ends up trying to load/store SIMD vectors to the stack a bunch of times.

Applying DLP to Matrix Multiply

- Each element of the product array is the result of dot product-ing two arrays.
- Would be useful to compute one element at a time; however, we can only load data that's consecutive in memory
- Therefore, we need to start by transposing the second matrix

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Applying DLP to Matrix Multiply

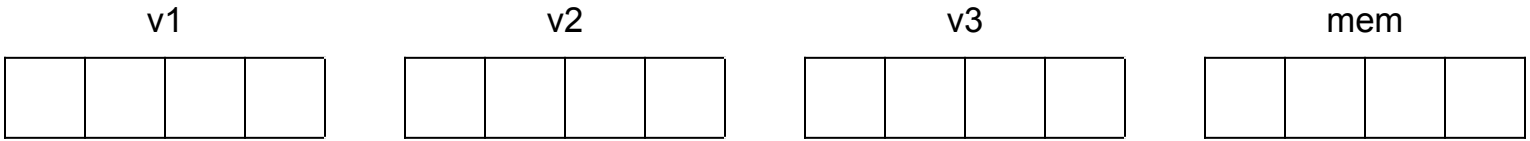
- How do we compute the dot product quickly?

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Applying DLP to Matrix Multiply

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

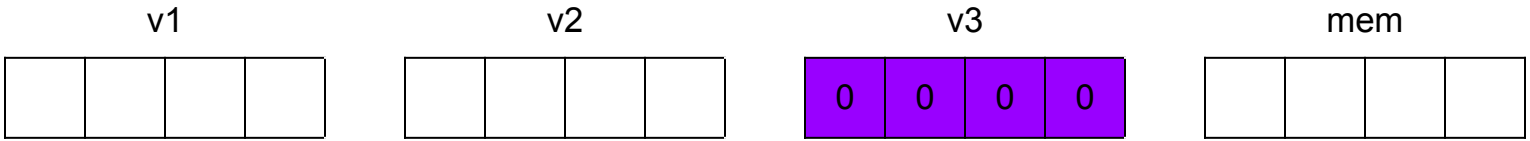


- Let's say we want to find the dot product of the two rows. How do we do this efficiently? (Assume the arrays are aligned)
- Step 1: Load 0s into v3:

```
__m128 v3 = _mm128_set1_ps(0);
```

Applying DLP to Matrix Multiply

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1



- Step 2: Load the first four numbers of each input into v1 and v2, respectively

```
__m128 v1 = _mm128_load_ps(&arr);
__m128 v2 = _mm128_load_ps(&arrtwo);
```

Applying DLP to Matrix Multiply

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

v1				v2				v3				mem			
1	2	3	4	1	1	1	1	0	0	0	0				

- Step 3: Multiply v1 and v2 together, and add that to v3
  - This procedure is so common, there's a single instruction to do this! (Well, for floats and doubles only)

```
v3 = _mm256_fmadd_ps(v1, v2, v3);
```

Applying DLP to Matrix Multiply

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

v1			
1	2	3	4

v2			
1	1	1	1

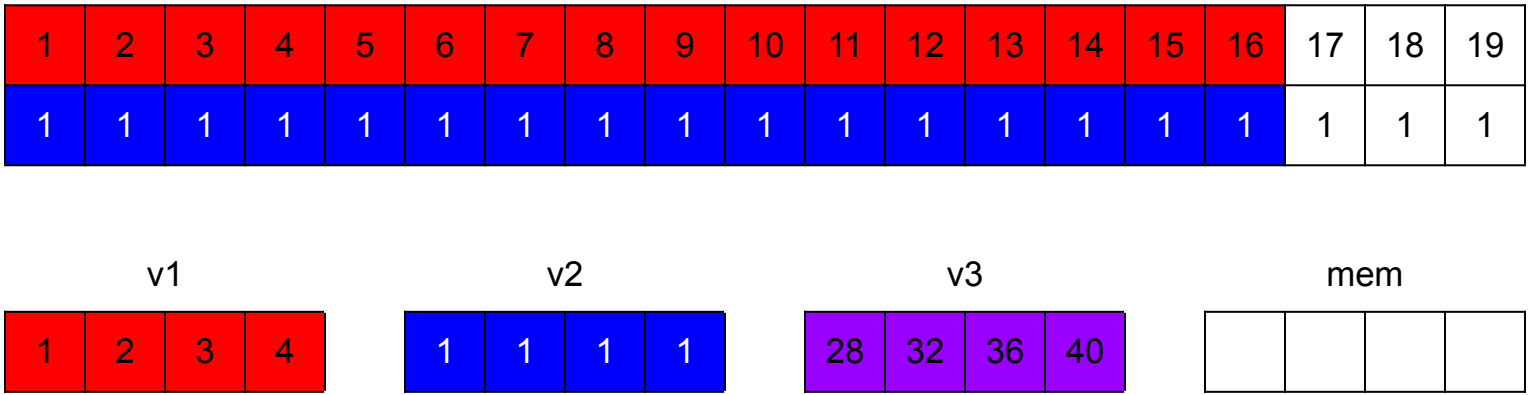
v3			
1	2	3	4

mem			

- Step 4: Repeat for the majority of the array

```
int i;
for(i = 0; i < arrlen/4*4;i+=4) {
    __m128 v1 = _mm128_load_ps(arr+i);
    __m128 v2 = _mm128_load_ps(arrtwo+i);
    v3 = _mm256_fmadd_ps(v1, v2, v3); }
```

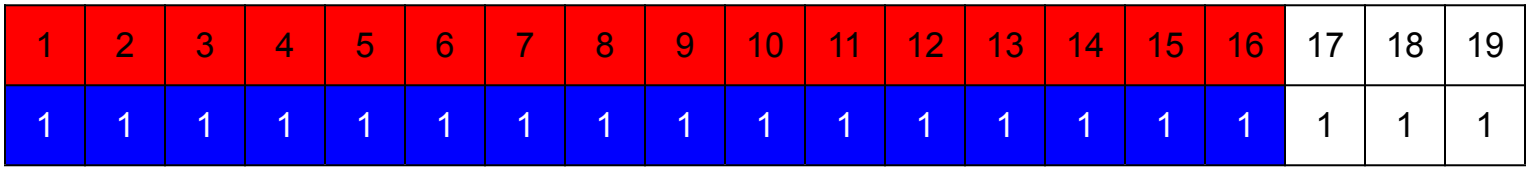
Applying DLP to Matrix Multiply

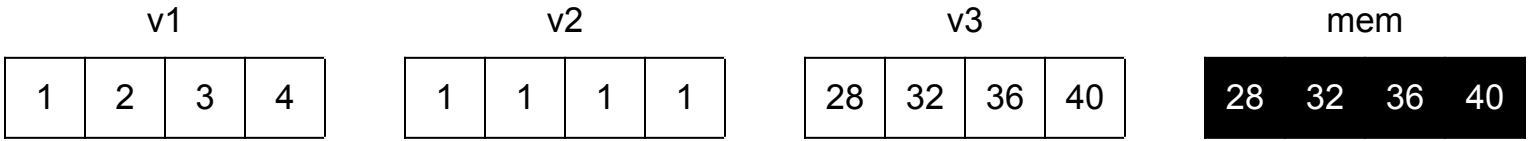


- Step 5: Store the results in memory somewhere.

```
//Force alignment
float mem[4] __attribute__ ((aligned (32)));
_mm128_store(mem, v3);
```

Applying DLP to Matrix Multiply

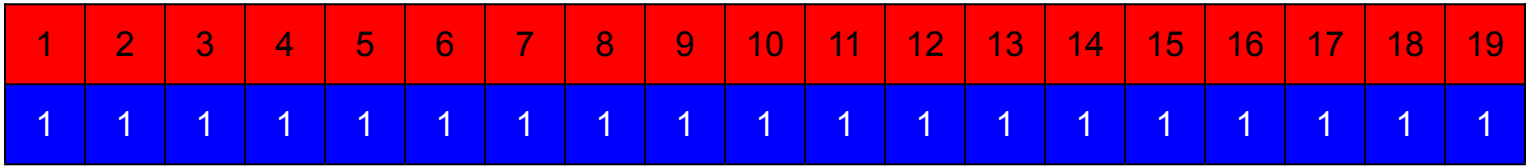


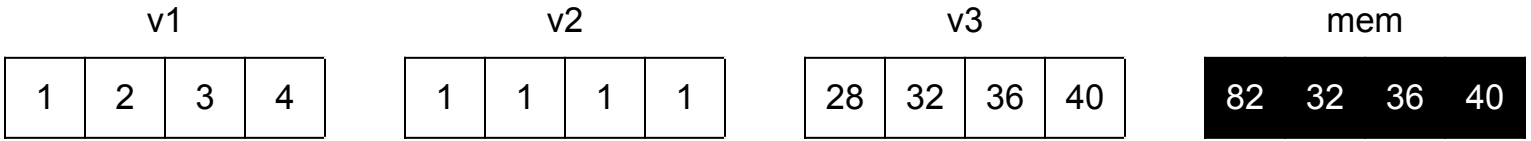


- Step 6: Resolve the tail case

```
for(;i<arrlen;i++) {
    mem[0]+=arr[i]*arrtwo[i];
}
```

Applying DLP to Matrix Multiply





- Step 7: Return the sum of mem

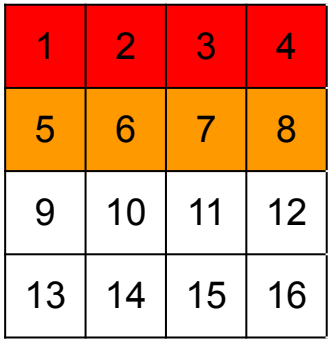
```
return mem[0]+mem[1]+mem[2]+mem[3];
```

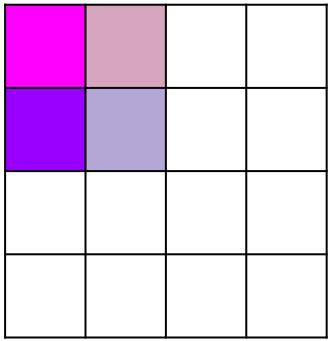
Applying DLP to Matrix Multiply


```
__m128 v3 = _mm128_set1_ps(0);
int i;
for(i = 0; i < arrlen/4*4;i+=4) {
    __m128 v1 = _mm128_load_ps(arr+i);
    __m128 v2 = _mm128_load_ps(arrtwo+i);
    v3 = _mm256_fmadd_ps(v1, v2, v3);
}
float mem[4] __attribute__ ((aligned (32)));
_mm128_store(mem, v3);
for(;i<arrlen;i++) {
    mem[0]+=arr[i]*arrtwo[i];
}
return mem[0]+mem[1]+mem[2]+mem[3];
```

Applying DLP to Matrix Multiply

- One final optimization here: right now, we load two rows to yield one value. Each row gets loaded n times.
- With enough vector registers, we can do this simultaneously to several cells at once
- This uses 8 vector registers, but computes 4 cells with 4 loads: 2x fewer loads!
- Does require an tail case for odd n





Conclusion

- Unfortunately, I couldn't get the code demo to work properly
  - SIMD version kept being slower. Either I'm bad, I don't have SIMD vectors on this computer, or gcc is already SIMDing it for me. Probably the first one.
- SIMD instructions are very useful when doing the same operation on a large array
- Keep in mind that loads and stores to SIMD registers take a long time, so the goal is to keep the data in registers for as long as possible; otherwise you spend most of your time in loads/stores