Vectorization

November 30, 2022

Principle

Vectorization consists in

- Reformulate algorithms
- So that they operate on **vectors**
  - Use only level-1 BLAS...

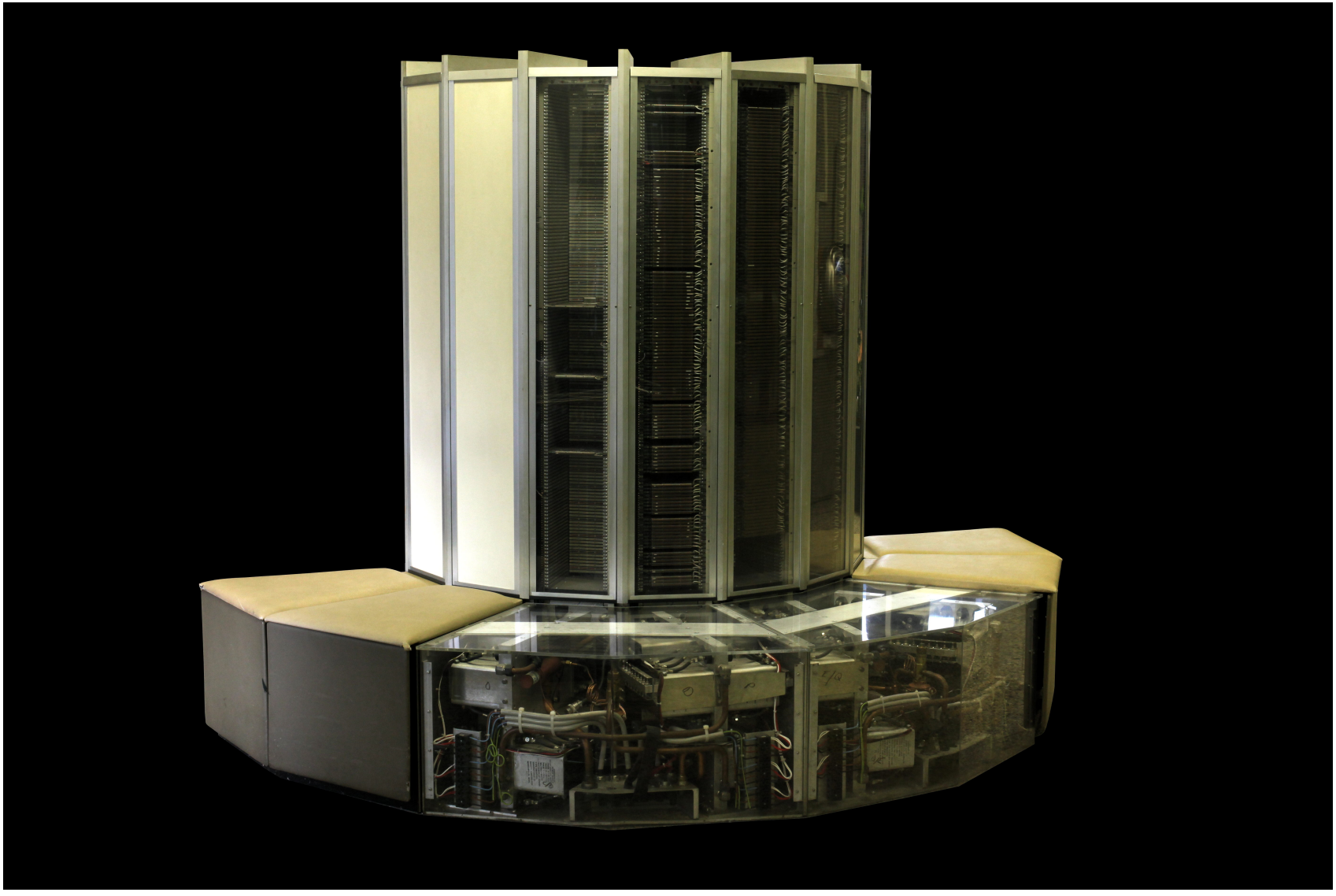
$A \leftarrow \lambda B + \mu C$	$A[i] \leftarrow \lambda B[i] + \mu C[i]$	
$A \leftarrow B + C \times D$	$A[i] \leftarrow B[i] + C[i] \times D[i]$	(*Fused Multiply-Add*)
$A \leftarrow \lvert B \rvert$	$A[i] \leftarrow \lvert B[i] \rvert$	
$A \leftarrow \max(B, C)$	$A[i] \leftarrow \max(B[i], C[i])$	
$A[B] \leftarrow C$	$A[B[i]] \leftarrow C[i]$	(*scatter*)
$A \leftarrow C[B]$	$A[i] \leftarrow C[B[i]]$	(*gather*)
$A \leftarrow B \leq C$	$A[i] \leftarrow B[i] \leq C[i]$	

Why?

Operations on vectors are intrinsically parallel

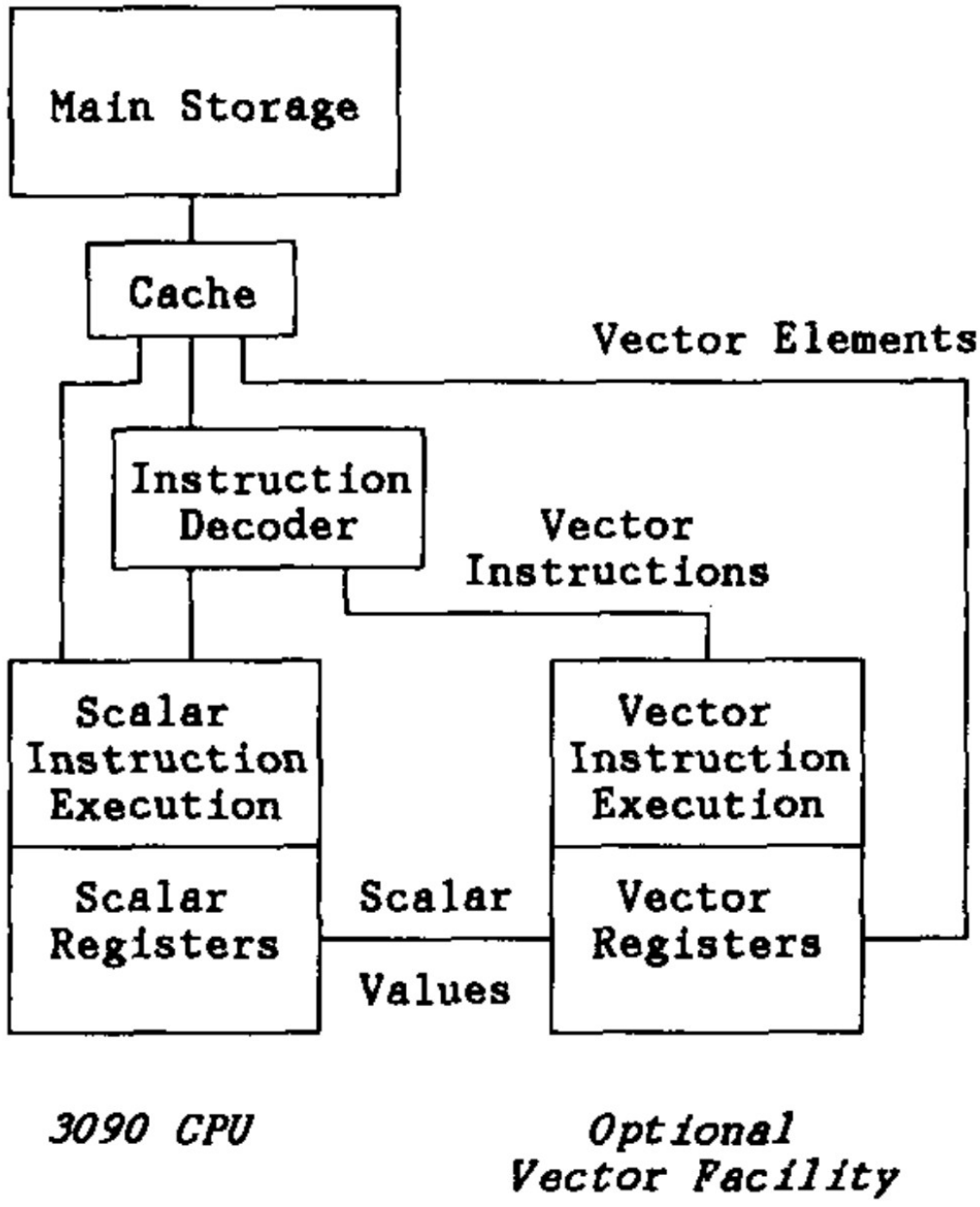
- Easy way to express parallelism
  - Fortran (≥ 90), `numpy` natively handle vectors
  - But not C...
- Vector processors used to rule the HPC world
  - Cray 1 (1975)
  - Cray X-MP (1982)
  - Cray 2 (1985)
  - IBM 3090 (1985)
  - Fujitsu VP2000 (1988)
  - NEC SX (1992)
  - ...

Vector Processors — Before



Vector Processors — Now

IBM 3090 CPU


Main Storage
Cache
Vector Elements
Instruction Decoder
Vector Instructions
Scalar Instruction Execution
Vector Instruction Execution
Scalar Registers
Scalar Values
Vector Registers
3090 CPU
Optional Vector Facility

Interest of Vector Instructions

- Possibility of parallel processing
  - Low end = a little
  - High-end = a lot
- A single instruction gives a lot of work
  - Save on instruction decoding
- Chained operations $\rightsquigarrow$ pipelining
  - Power efficiency

Why did it disappear?

- Needs **fast memory** to transfer the vectors
- Scatter/gather very hard on memory subsystem

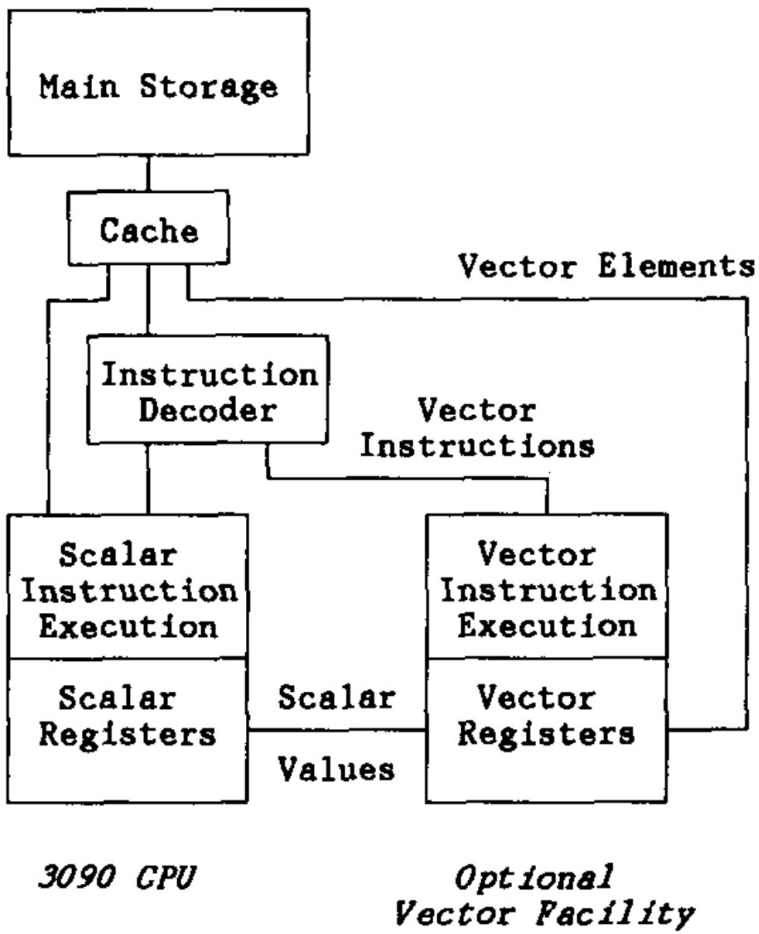
$\Rightarrow$ (expensive) processors/machines *dedicated* to HPC

- But trend towards *commodity components* (ASCI red)

Present Day: SIMD Instructions

1985 IBM 3090, vector registers of $32Z$ bits

- Z unspecified, *implementation-dependent*
- Concretely, $Z = 128 \rightsquigarrow$ 4096-bit vectors



Present Day: SIMD Instructions

1985 IBM 3090, vector registers of $32Z$ bits

- Z unspecified, *implementation-dependent*
- Concretely, $Z = 128 \rightsquigarrow$ 4096-bit vectors

Present Day: SIMD Instructions

1985 IBM 3090, vector registers of $32Z$ bits

- Z unspecified, *implementation-dependent*
- Concretely, $Z = 128 \rightsquigarrow$ 4096-bit vectors

1996 Intel Pentium MMX: 64 bit-vectors (`8*int8/4*int16`)

Present Day: SIMD Instructions

1985 IBM 3090, vector registers of $32Z$ bits

- Z unspecified, *implementation-dependent*
- Concretely, $Z = 128 \rightsquigarrow$ 4096-bit vectors

1996 Intel Pentium MMX: 64 bit-vectors (`8*int8/4*int16`)

1999 Intel SSE: 128-bit vectors, `4*float` only

2001 Intel SSE2: more types, `int` and `double`

2004 Intel SSE3: more operations

2007 Intel SSE4: more operations

2010 Intel AVX: 256-bit vectors, `8*float` only

2010 Intel AVX2: more types, `int` and `double` + `gather`

2017 Intel AVX512: 512-bit, all types + `scatter/gather/mask`

ISA	Accronym	Name	Year	# regs	|regs|
IBM 3090		Vector Facility	1985	16	variable
x86-64	MMX	Multi Media eXtensions	1996	8	64
	3DNow!		1998	8	64
	SSE SSE2 SSE3 SSSE3 SSE4	Streaming SIMD Extensions	1999 2001 2004 2004 2007	16	128
	AVX AVX2	Advanced Vector eXtensions	2010 2013	16	256
	AVX512		2017	32	512
PowerPC	Altivec		1999	32	128
ARM v7	NEON	Advanced SIMD extension	2005 2011	16 32	128
ARM v8	SVE	Scalable Vector Extensions	2020	32	variable

- ARMv8 SVE : registers of $(128Z)$-bit ($1 \leq Z \leq 16$)
  - `fugaku` has SVE with 512-bit registers
  - `AWS graviton` has SVE with 256-bit registers
- `RISC-V` has an "old school" vector extension (IBM 3090-like)

Two Styles

"Real" (old-school) vector architectures

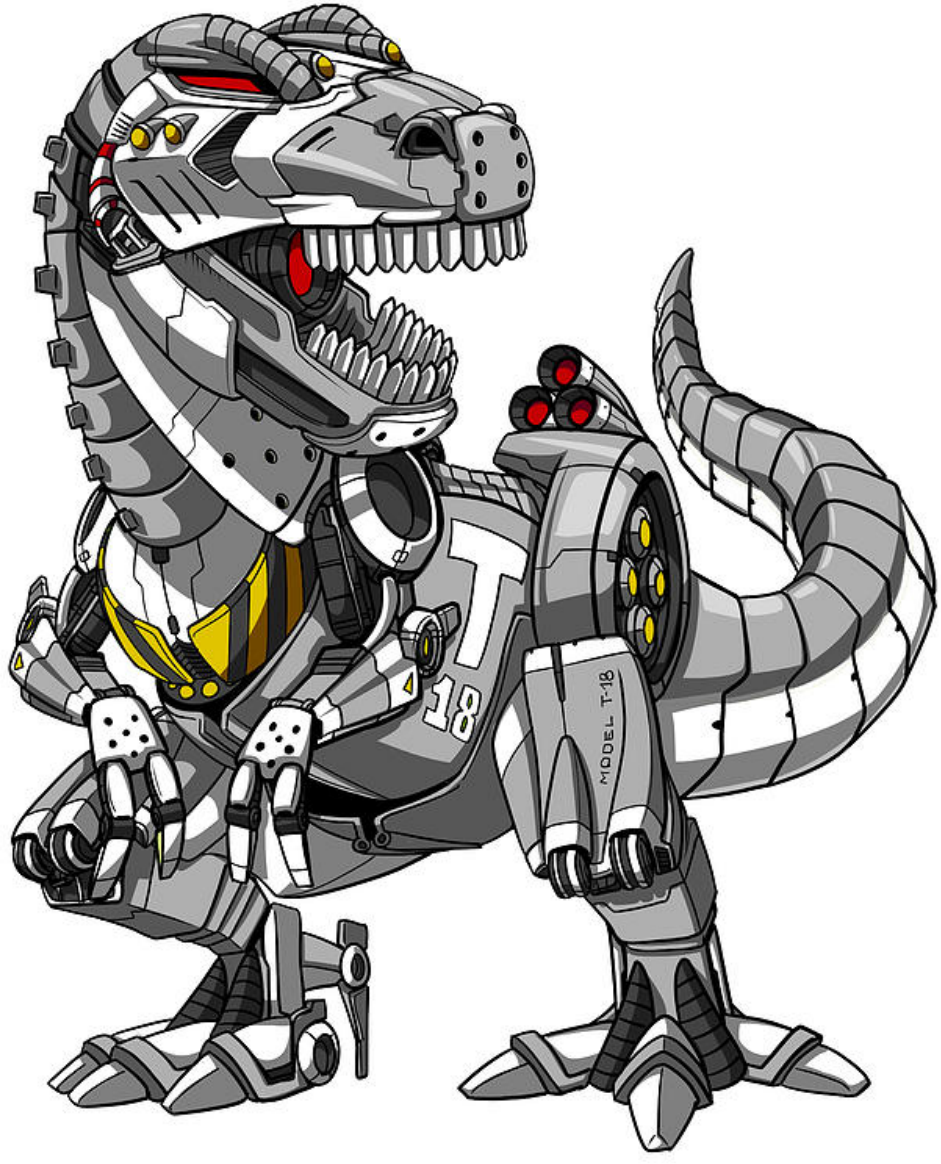
- Unknown register size, or no registers at all
- Functionality: Scatter + Gather + masks
- **Revival**: ARMv8 SVE, RISC-V, AVX-512, ...

SIMD Instructions available in consumer CPUs

- Fixed register size
- Stacking of successive generations
- Constrained memory access

In between / besides: GPUs

Vector Processors — Revival?



- ARM Neon is there in all smartphones
  - And recent Apple macbooks
- Nearly all x86 CPUs presently have AVX2
- AVX512 is not for everybody
  - Intel Xeon bronze/silver/gold (since 2017)
  - Intel Some core i7/i9 ("X" or "extreme")
  - AMD EPYC "Zen 4"
- AVX512 @ g5k:
  - dahu (Grenoble)
  - gros (Nancy)

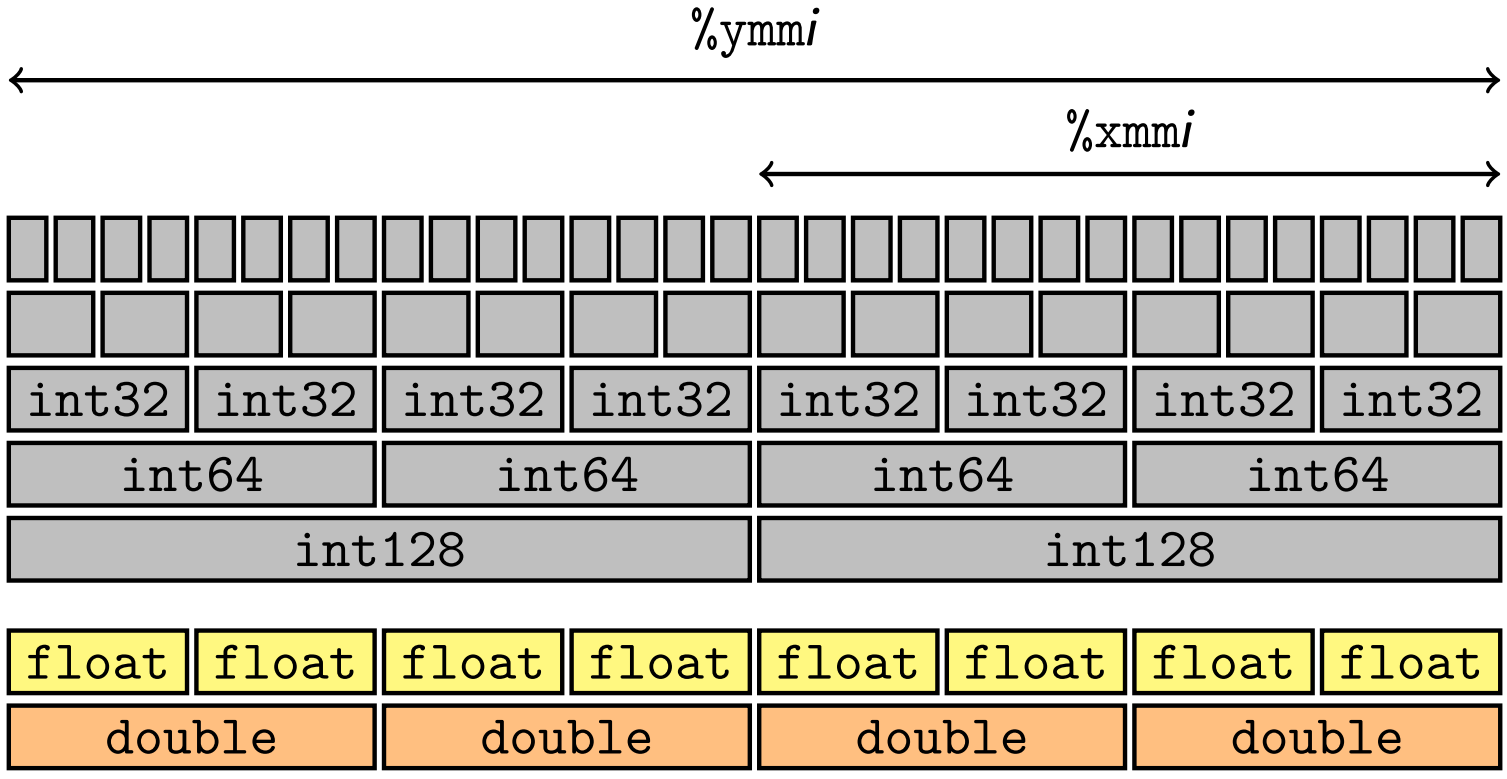
```
chbouillaguet@gros-70:~$ cat /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3
invpcid_single pti intel_ppin ssbd mba ibrs ibpb stibp tpr_shadow vnmi
flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2
erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap
clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm
ida arat pln pts pku ospke flush_l1d
```

Problems with SIMD Instruction Sets

Their multiplicity makes them hard to use

- Needs recompilation at each new iteration
  - Even rewrite code / change algorithms...
- Makes code non-portable
  - Does the CPU that will run it have AVX-xxx?
  - Query the CPU at runtime...
  - ... and choose appropriate function (sigh)

SSEx + AVX + AVX2



Instructions	registers	size
SSE*x*	%xmm0, ..., %xmm15	128-bit
AVX + AVX2	%ymm0, ..., %ymm15	256-bit
AVX512	%zmm0, ..., %zmm31 %k0, ..., %k7	512-bit 64-bit

SIMD Instructions Nomenclature

Floating-point SIMD instructions have compound names:

`<prefix><op><simd or not><type>`

where

- `<prefix>`: empty (SSE), `v` (AVX, AVX2, AVX512)
- `<op>` `add`, `sub`, `mul`, `div`, `fmadd`, `min`, `max`, `abs`, `floor`, `ceil`, `round`, ...
- `<simd or not>` `s` (*scalar*), `p` (*packed* — i.e. vector).
- `<type>` `s` (*single*, `float`), `d` (`double`)

E.g. `vmaxpd`, `vmulss`, etc.

Practical Vectorization

General Principle

1. Load data into vector registers
2. Use vector instructions
3. Write result back to memory

(minimise these transfers is an important goal)

Use cases

- Happens **inside a single thread**
  - Parallelization = distribute over several cores
  - Vectorization = use SIMD instructions inside **ONE** core
- Only **small code chunks** can (usually) be vectorized
  - Tight **loops** (limits of the instruction set)

  $\Rightarrow$ Target the **innermost** loops

Obstacles

- Data dependencies (dependent iterations)

```
for (int i = 1; i < n; i++)
    a[i] += a[i-1];
```

  - No possible parallel processing

↪ Must change the algorithm

- Conditional instructions (`if`)

```
for (int i = 1; i < n; i++)
    if (a[i] != 0)
        a[i] = 1 / a[i];
```

  - Potentially manageable (overhead — to avoid if possible)

- Complex memory access pattern and/or alignment problems
  - Loading **contiguous** array slices only

- Complex operations (sin, cos, log, exp, my_function, ...)

Data Alignment

Rule of thumb

xxx-bytes vector must be located at an address multiple of *xxx*

Solutions

- `double A[n] __attribute__ ((aligned(32)));`
  - This is valid C code
- `int posix_memalign(void **ptr, size_t align, size_t size);`
- `void *aligned_alloc(size_t alignment, size_t size);`
  - Included in C11

Complex memory access pattern

Array of struct

```
struct point {
      double x, y, z;
};

struct point *A;
```

xyzxyzxyzxyzxyzxyz....

Struct of Array

```
struct points {
      double *x, *y, *z;
};
struct points A;          // facilitated batch processing
                          // arrays of uniform elements
```

xxxxxxxxxxxxxx....

yyyyyyyyyyyyyy....

zzzzzzzzzzzzzz....

Vectorizing Simple Loops: *strip-mining*

Original code

```
for (int i = 0; i < n; i++)
    u[i] = u[i] + alpha * v[i];
```

Transformed code

```
int m = n - (n % 4);
/* Processing in batches of 4 with SIMD instructions */
for (i = 0; i != m; i += 4) {
    u[i + 0] = u[i + 0] + alpha * v[i + 0];
    u[i + 1] = u[i + 1] + alpha * v[i + 1];
    u[i + 2] = u[i + 2] + alpha * v[i + 2];
    u[i + 3] = u[i + 3] + alpha * v[i + 3];
}
/* epilogue when n is not a multiple of 4 */
for (int i = m; i < n; i ++)
    u[i] = u[i] + alpha * v[i];
```

- Potentially: prologue to ensure alignment
- The epilogue can be avoided with **padding**

strip-mining?

Algorithmic Modifications

Starting Point

```
double sum = 0;
for (int i = 0; i < n; i++)
    sum += a[i];                    // data dependency (sum)
```

Transformed code

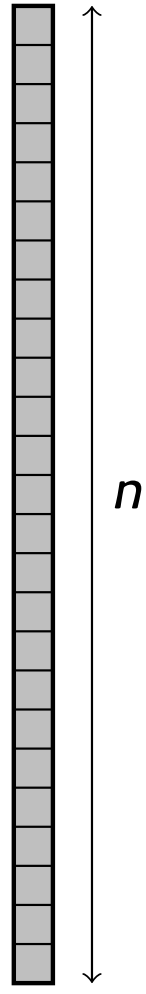
```
double __sum[4] = {0, 0, 0, 0};
for (int i = 0; i < n; i += 4) {        // assume that n is multiple of 4
    __sum[0] += a[i + 0];
    __sum[1] += a[i + 1];
    __sum[2] += a[i + 2];
    __sum[3] += a[i + 3];
}
// final reduction (non-vectorized)
double sum = __sum[0] + __sum[1] + __sum[2] + __sum[3];
```

- Loop is now vectorizable
- Small overhead in epilogue

Apply a Full Sequential Algorithm on Vectors

Sequential Algorithm $\mathcal{A}$

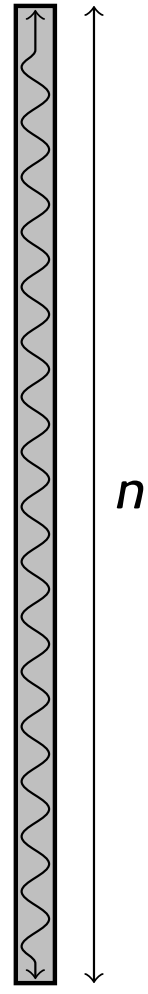
- Applies to an array X
- *"data-oblivious"*: sequence id instructions independent of the values



Apply a Full Sequential Algorithm on Vectors

Sequential Algorithm $\mathcal{A}$

- Applies to an array X
- *"data-oblivious"*: sequence id instructions independent of the values



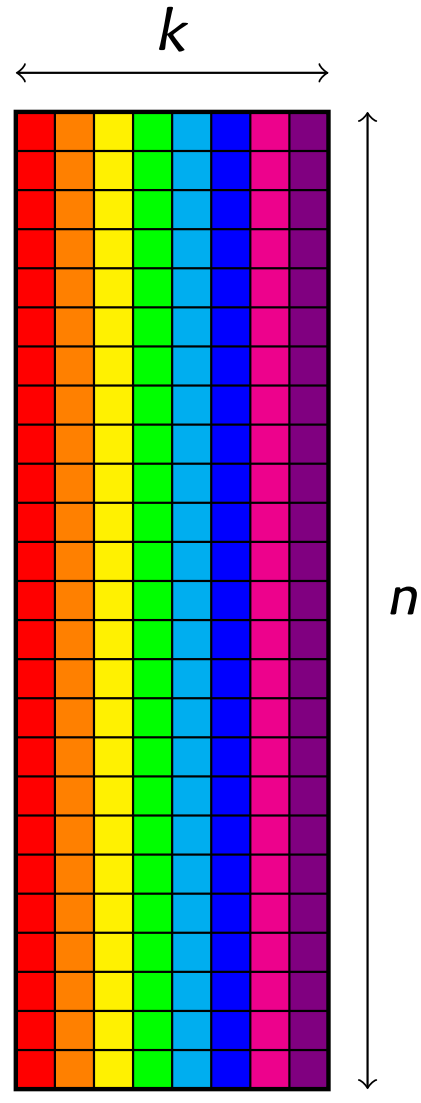
Apply a Full Sequential Algorithm on Vectors

Sequential Algorithm $\mathcal{A}$

- Applies to an array X
- *"data-oblivious"*: sequence id instructions independent of the values

Apply k times the algorithm in parallel

- scalars $\rightarrow$ vectors of size k
- scalar operations $\rightarrow$ SIMD



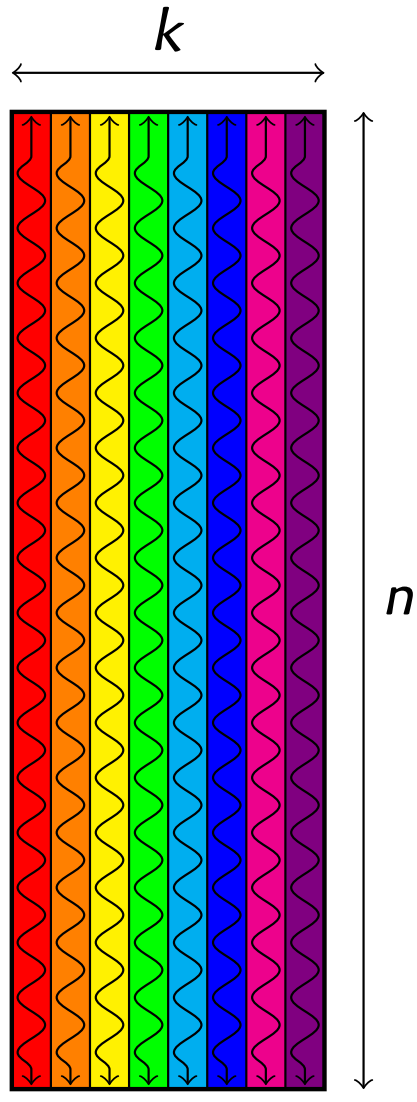
Apply a Full Sequential Algorithm on Vectors

Sequential Algorithm $\mathcal{A}$

- Applies to an array X
- *"data-oblivious"*: sequence id instructions independent of the values

Apply k times the algorithm in parallel

- scalars $\rightarrow$ vectors of size k
- scalar operations $\rightarrow$ SIMD



How to Use SIMD / Vector Instructions?

1. ~~Write assembler code directly~~
2. Use *"instrinsics"* offered by compilers
3. Let the compiler do it (automatic vectorization)
4. Provide vectorization directives (OpenMP $\geq$ v4.0)
5. Use `gcc`-specific vector extensions

Using the *Intrinsics*

- Introduced by Intel, later adopted by other compilers
- Pseudo-functions that emit **one** SIMD instructions
- Need to know the instruction set(s)

Headers

```
#include <xmmintrin.h>          // SSE
#include <emmintrin.h>          // SSE2
#include <pmmintrin.h>          // SSE3
#include <tmmintrin.h>          // SSSE3
#include <smmintrin.h>          // SSE4.1
#include <nmmintrin.h>          // SSE4.2
#include <immintrin.h>          // AVX & AVX2 & AVX512
```

Data Types for the *Intrinsics*

Match the hardware "vectors"

`__m512`	512 bits (16 `float`)
`__m512d`	512 bits (8 `double`)
`__m512i`	512 bits (ints)
	+ `__mmask8`, `__mmask16`, `__mmask32`, `__mmask64`
`__m256`	256 bits (8 `float`)
`__m256d`	256 bits (4 `double`)
`__m256i`	256 bits (ints)
`__m128`	128 bits (4 `float`)
`__m128d`	128 bits (2 `double`)
`__m128i`	128 bits (ints)

Examples (AVX2)

- `_mm256_set1_ps`
- `_mm256_load_pd`
- `_mm256_xor_si256`
- `_mm256_srli_epi32`
- `_mm256_i32gather_mask_pd`
- `_mm256_cmpeq_epi32`
- `_mm256_movemask_epi8`
- ...

General Nomenclature

```
   _mm_<operation>_<suffix>(param1, param2)
_mm256_<operation>_<suffix>(param1, param2, param3)
_mm512_<operation>_<suffix>(param1, param2, param3)
```

Two-part `<suffix>`

- 1st part
  - `p` (packed) and `ep` (extended packed): SIMD
  - `s` (scalar): only 1st vector element is active
- 2nd part (type)
  - `[s/d]`: **s**ingle- or **d**ouble-precision float
  - `[i/u]nnn`: s**i**gned or **u**nsigned `nnn`-bits integer (`nnn` is 512, 256, 128, 64, 32, 16, or 8)

Example

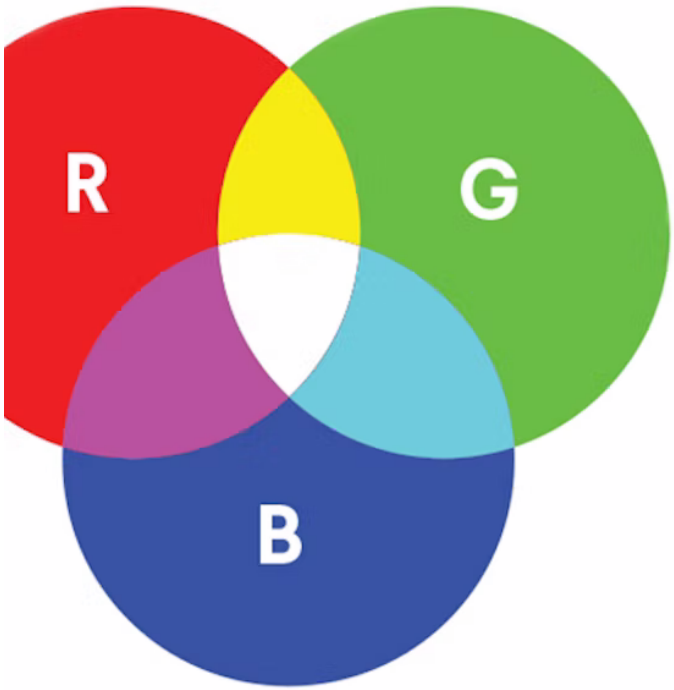
Original Code

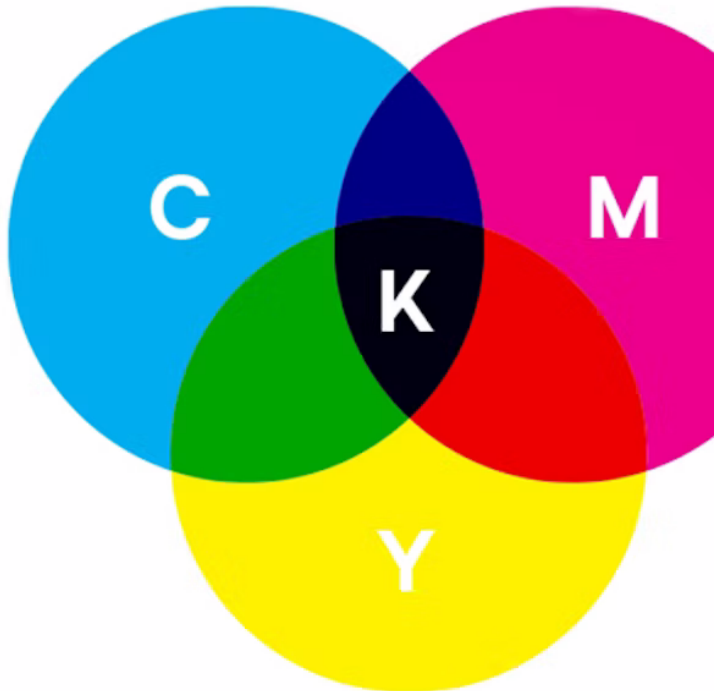
```
double alpha;
double u[], v[];
// ...
for (int i = 0; i < n; i++)
    u[i] = u[i] + alpha * v[i];
```

Vectorized code (AVX2)

```
__m256d valpha = _mm256_set1_pd(alpha);    // (alpha, alpha, alpha, alpha)
for (i = 0; i != m; i += 4) {
    /* processing by batches of 4 */
    __m256d vu = _mm256_load_pd(&u[i]);          // alignment required
    __m256d vv = _mm256_load_pd(&v[i]);          // alignment required
    __m256d vres = _mm256_fmadd_pd(valpha, vv, vu);
    _mm256_store_pd(&u[i], vres);                // alignment required
}
```

Non-Trivial Example: Colorimetric Conversions





Non-Trivial Example: Colorimetric Conversions

R, G, B, C, M, Y, K = floats in $[0; 1]$

Easy Direction

$$\begin{cases} R &= (1-C) \times (1-K) \\ G &= (1-M) \times (1-K) \\ B &= (1-Y) \times (1-K) \end{cases}$$

Problematic direction

- If $(R, G, B) = (0, 0, 0)$ then $(C, M, Y, K) = (0, 0, 0, 1)$
- Otherwise

$$\begin{cases} K &= 1 - \max(R, G, B) \\ C &= 1 - R/(1-K) \\ M &= 1 - G/(1-K) \\ Y &= 1 - B/(1-K) \end{cases}$$

Problems with Conditional Instructions

Example: RGB → CMYK

$K = 1 - \max(R, G, B)$

```
__m256 vone = _mm256_set1_ps(1.0);                    // (1, 1, 1, 1, 1, 1, 1, 1)
// ...
/* process a chunk of 8 colors */
__m256 vR = _mm256_load_ps(&R[i]);                       // alignment required
__m256 vG = _mm256_load_ps(&G[i]);                       // alignment required
__m256 vB = _mm256_load_ps(&B[i]);                       // alignment required
__m256 vmax = _mm256_max_ps(_mm256_max_ps(vR, vG), vB);
__m256 vK = _mm256_sub_ps(vone, vmax);
_mm256_store_ps(&K[i], vk);                              // alignment required
```

If $K < 1$

```
__m256 vC = _mm256_sub_ps(vone, _mm256_div_ps(vR, vmax));
__m256 vM = _mm256_sub_ps(vone, _mm256_div_ps(vG, vmax));
__m256 vY = _mm256_sub_ps(vone, _mm256_div_ps(vB, vmax));
```

Problems with Conditional Instructions

Example: RGB → CMYK

If $K = 1$

```
__m256 vzero = __m256 _mm256_setzero_ps();
__m256 vmask = _mm256_cmp_ps(vK, vone, _CMP_EQ_OQ);
// vmask[i] == (vK[i] == 1.0) ? 0xffffffff : 0x00000000
vC = _mm256_blendv_ps(vzero, vC, vmask);
vM = _mm256_blendv_ps(vzero, vM, vmask);
vY = _mm256_blendv_ps(vzero, vY, vmask);
// blendv : vX[i] =     vmask[i]   ? vzero[i] : vX[i]
//                  (vK[i] == 1.0) ? vzero[i] : vX[i]
```

Intructions for shuffling data inside registers:

- `blend`, `blendv`, `broadcast`, `extract`, `insert`, `permute`, `permutevar`, `shuffle`, `unpackhi`, `unpacklo`, ...

Problems with Conditional Instructions

Example: RGB → CMYK

```
// R, G, B, C, M, Y, K must be aligned
__m256 vone, vzero, vmask, vR, vG, vB, vC, vM, vY, vK;
vone = _mm256_set1_ps(1.0);              // (1, 1, 1, 1, 1, 1, 1, 1)
vzero = _mm256_setzero_ps();             // (0, 0, 0, 0, 0, 0, 0, 0)
for (int i = 0; i < n; i += 8) {
    vR = _mm256_load_ps(&R[i]);
    vG = _mm256_load_ps(&G[i]);
    vB = _mm256_load_ps(&B[i]);
    vmax = _mm256_max_ps(_mm256_max_ps(vR, vG), vB);
    vK = _mm256_sub_ps(vone, vmax);
    vC = _mm256_sub_ps(vone, _mm256_div_ps(vR, vmax));
    vM = _mm256_sub_ps(vone, _mm256_div_ps(vG, vmax));
    vY = _mm256_sub_ps(vone, _mm256_div_ps(vB, vmax));
    vmask = _mm256_cmp_ps(vK, vone, _CMP_EQ_OQ);
    vC = _mm256_blendv_ps(vzero, vC, vmask);
    vM = _mm256_blendv_ps(vzero, vM, vmask);
    vY = _mm256_blendv_ps(vzero, vY, vmask);
    _mm256_store_ps(&C[i], vC);
    _mm256_store_ps(&M[i], vM);
    _mm256_store_ps(&Y[i], vY);
    _mm256_store_ps(&K[i], vK);
}
```

OpenMP `simd` Directive

OpenMP reminder

- requires compiling with `-fopenmp`
- `-fopenmp-simd` enables SIMD but not threads

```
#pragma omp simd
for (int i = 0; i < n; i++)
   ...
```

- Groupe iterations in *chunks* of size *N*
  - Do the computation using *N*-wide SIMD instructions
- We “promise” the compiler that the loop is vectorizable
- No multi-thread parallelization there

OpenMP `simd` Directive

```
#pragma omp simd [clause[[,] clause],...]
for (int i = 0; i < n; i++)
    ...
```

Possible clauses (non-exhaustive list)

- `reduction(+:v)`: already known
  - Breaks data dependency on accumulator variable
- `simdlen(length)`: desired *chunk* size
- `safelen(length)`: maximum allowed *chunk* size
  - E.g., beyond this a dependency problem could occur
- `aligned(v[:k])`: promise that *v* is aligned on `k` bytes
- `linear(x[:step])`: *x* is an affine function of *i*
  (`x_i = x_init + i * step;`)

Automatic Vectorization

- Fortran a has explicit vectors
  - Easy to vectorize
- In C, compiler detects **vectorizable loops**

C Code

```
for (int i = 0; i < n; i++)
        C[i] = lambda * A[i] + mu * B[i];
```

Assembler code produced by gcc

```
.loop:
        vmulpd   (%rsi,%rax), %ymm3, %ymm0
        vmulpd   (%rcx,%rax), %ymm2, %ymm1
        vaddpd   %ymm1, %ymm0, %ymm0
        vmovapd  %ymm0, (%rdx,%rax)
        addq     $32, %rax
        cmpq     %rcx, %rax
        jne      .loop
```

SIMD Instructions in gcc

- **automatic vectorization** is disabled by default
  - Without explicit options, "optimization level" = 0
  - automatic vectorization (and OpenMP) disabled
- Enable with option `-ftree-vectorize`
  - `-O3` activates this feature
- We're doing HPC: always compile with `-O2` or `-O3`

```
~$ gcc -O1 -ftree-vectorize vectorisation_auto.c
~$ gcc -O1 -fopenmp vectorisation_omp.c
```

- `gcc` does **not** emit AVX2 (resp. FMA) instructions by default
  - Add `-mavx2` (resp. `-mfma`) to the command-line
- Verbosity and diagnostics:
  - `-fopt-info-vec`, `-fopt-info-vec-missed` and `-fopt-info-vec-all`

Case Study

Discrete Fourier Transform (DFT)

- The DFT of an array X of n (complexes) numbers is

$$Y[k] = \sum_{j=0}^{n-1} X[j]\omega_n^{jk}, \qquad \text{with} \qquad \omega_n = e^{-\frac{2i\pi}{n}} \qquad (0 \leq k < n)$$

- Optimized libraries (e.g. FFTW)

Rewriting the definition

When $n = n_1 \times n_2$, we set $j = j_1 n_2 + j_2$ and $k = k_1 + k_2 n_1$

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

FFT: Recursive Algorithm

$n = n_1 \times n_2$, we set $j = j_1 n_2 + j_2$ et $k = k_1 + k_2 n_1$

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

Algorithm to compute the DFT of size $n_1 \times n_2$

1. Do n_2 DFTs of size n_1 (internal sum)
2. Multiply by the *twiddle factors* ${\omega_n}^{j_2 k_1}$
3. Do n_1 DFTs of size n_2 (external sum)

FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

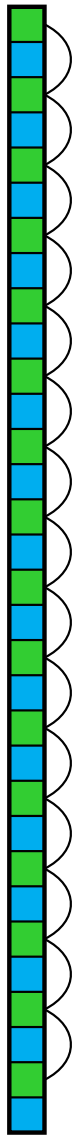
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```

FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```

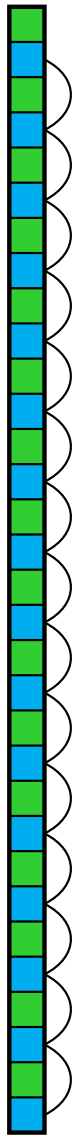


FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```



FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```

FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;    // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```

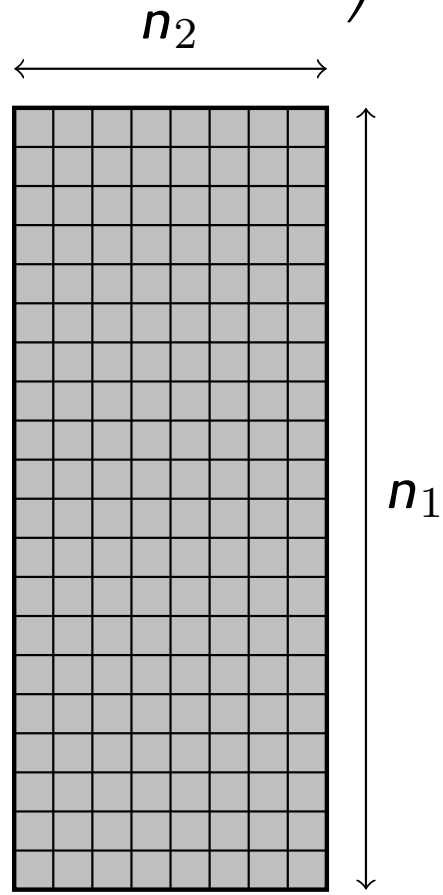
FFT: Recursive Algorithm

Classical Presentation

- Common choice : $n_2 = 2$
  - *"Radix-2 Decimation in Time"*
- FFT of size 2 : $(x, y) \rightarrow (x + y, x - y)$

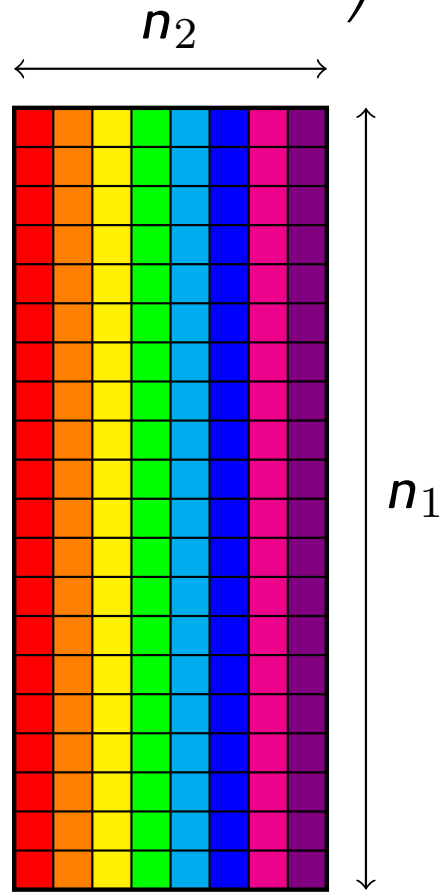
```
void FFT(const double * X, double *Y, int n, int s)
{
    if (n == 1) {
       Y[0] = X[0];
       return;
    }
    double omega_n = exp(-2*I*pi / n);
    double omega = 1;   // twiddle factor
    FFT(&X[0], &Y[0]  , n/2, 2*s);
    FFT(&X[s], &Y[n/2], n/2, 2*s);
    for (int i = 0; i < n/2; i++) {
        double p = Y[i];
        double q = Y[i + n/2] * omega;
        Y[i] = p + q;
        Y[i + n/2] = p - q;
        omega *= omega_n;
    }
}
```

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$



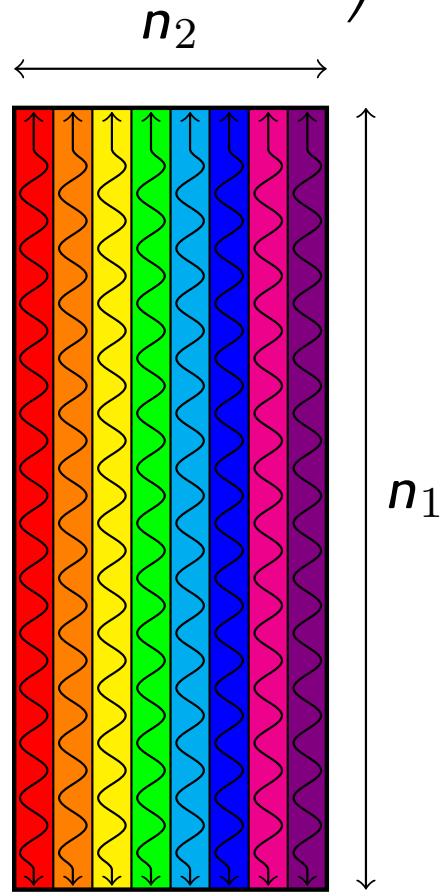
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$



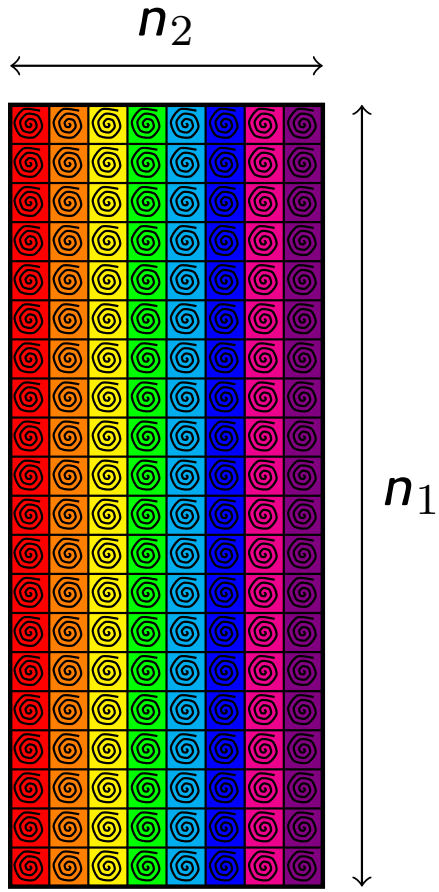
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] {\omega_{n_1}}^{j_1 k_1} \right) {\omega_n}^{j_2 k_1} \right] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$



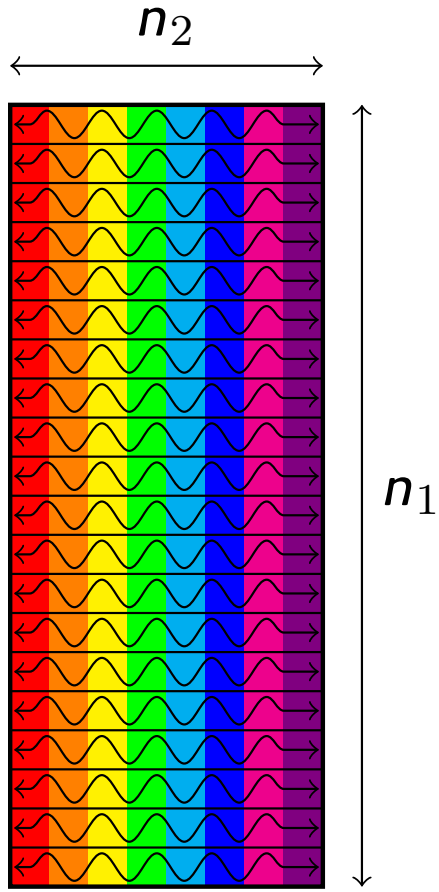
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} \left(U[k_1 n_2 + j_2] \omega_n^{j_2 k_1} \right) \omega_{n_2}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot \omega_n^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$



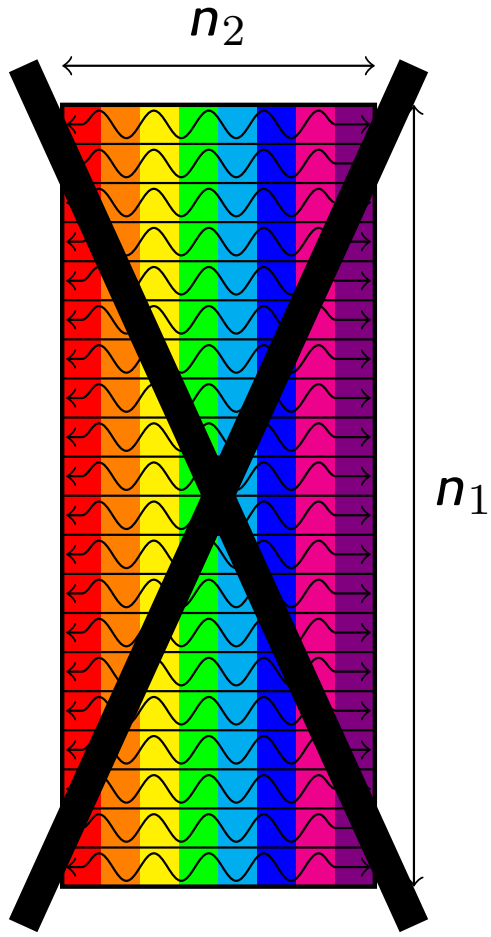
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- $Y[k_1, \star] \leftarrow FFT(V[k_1, \star])$
  - $0 \leq k_1 < n_1$



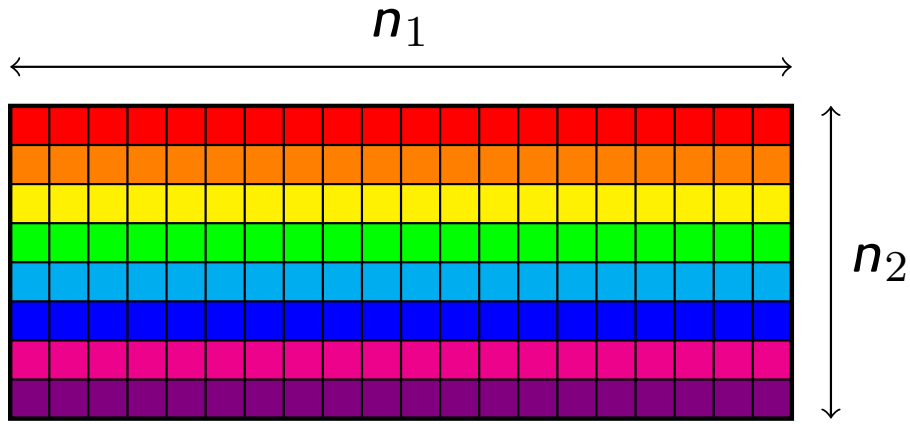
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- $Y[k_1, \star] \leftarrow FFT(V[k_1, \star])$
  - $0 \leq k_1 < n_1$



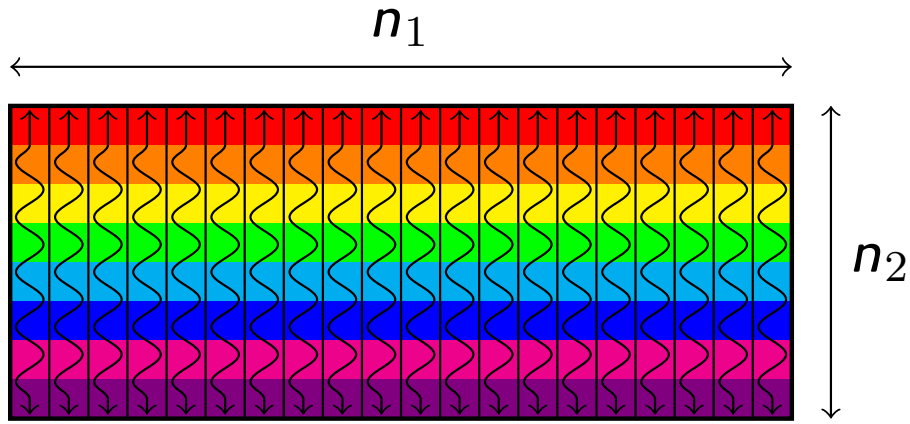
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- Transpose V
  - overhead from vectorization



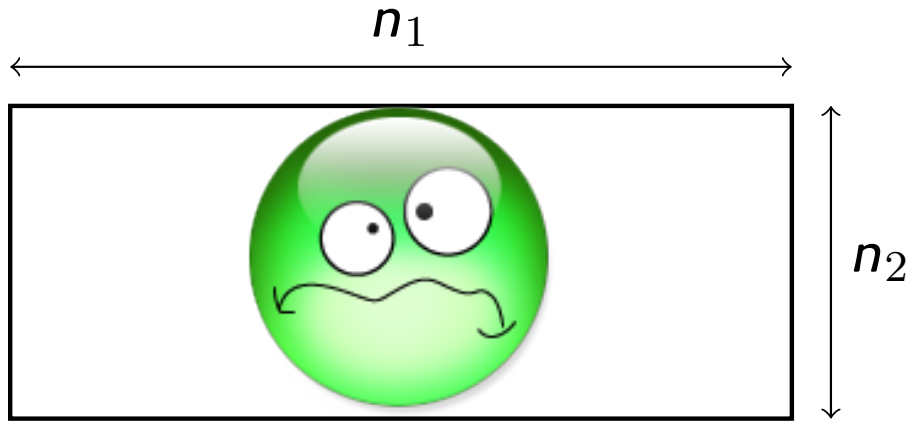
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- Transpose V
  - overhead from vectorization
- $Y[\star, k_1] \leftarrow FFT(V[\star, k_1])$
  - $0 \leq k_1 < n_1$



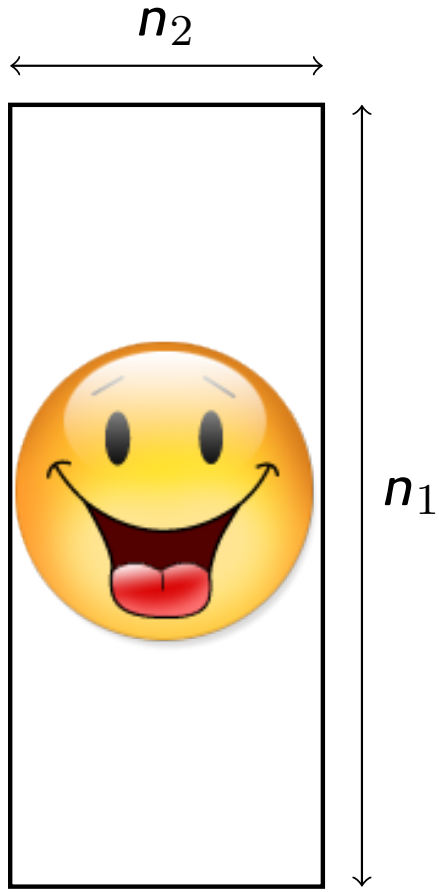
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- Transpose V
  - overhead from vectorization
- $Y[\star, k_1] \leftarrow FFT(V[\star, k_1])$
  - $0 \leq k_1 < n_1$

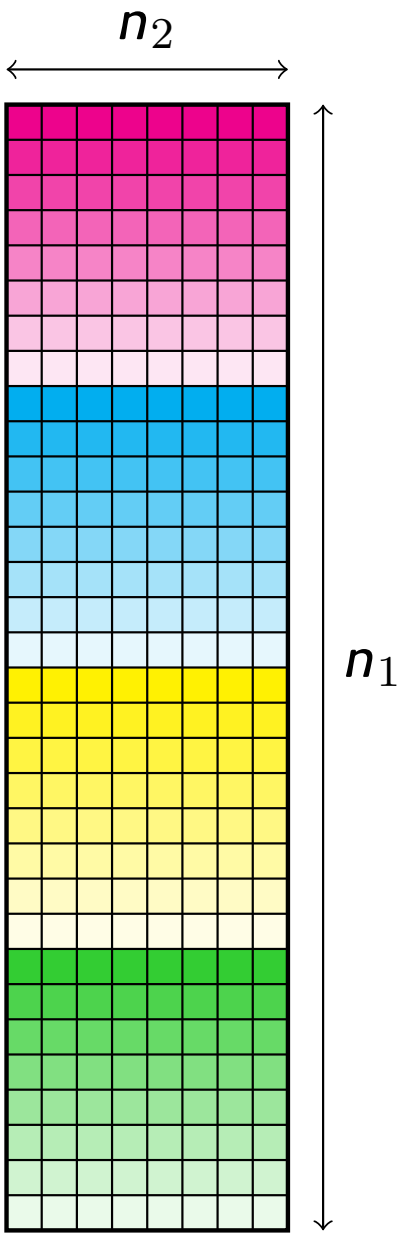


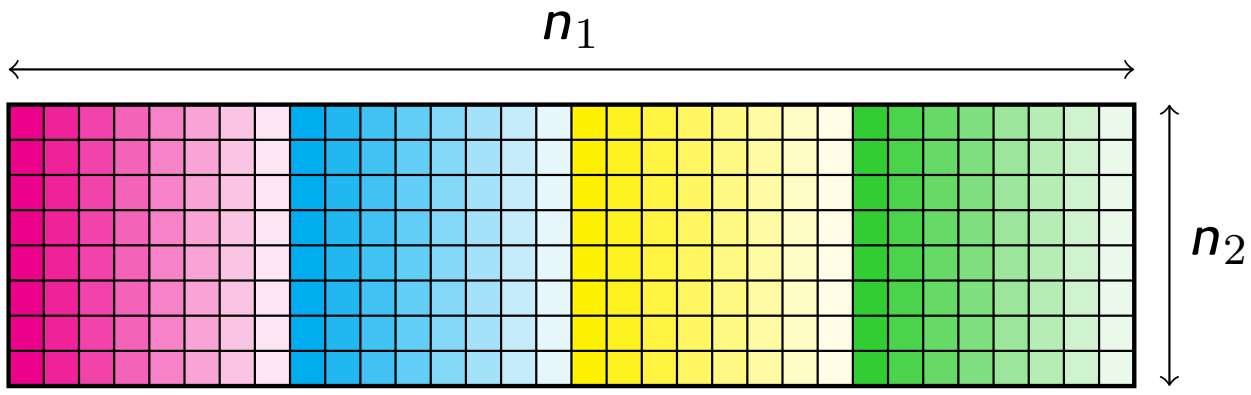
$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1} V[k_1 n_2 + j_2] {\omega_{n_2}}^{j_2 k_2}$$

- $U[\star, j_2] \leftarrow FFT(X[\star, j_2])$
  - $0 \leq j_2 < n_2$
- $V[k_1, j_2] \leftarrow U[k_1, j_2] \cdot {\omega_n}^{j_2 k_1}$
  - $0 \leq j_2 < n_2$
  - $0 \leq k_1 < n_1$
- Transpose V
  - overhead from vectorization
- $Y[\star, k_1] \leftarrow FFT(V[\star, k_1])$
  - $0 \leq k_1 < n_1$
- Retranspose Y
  - overhead from vectorization



Addendum: Vectorized Transposition





"Fun" exercise

Write code that transpose

- A 4×4 matrix of `double`
- A 8×8 matrix of `float`
- Using *intrinsics*