

# Optimization

**Giannis Koutsou**

CDS-1, Stimulate Workshop,  
November 19 - December 7, 2018,  
Jülich Supercomputing Center



# Outline

## What to expect from this course

- A look at some example-driven, hands-on techniques for optimizing computational kernels
- The techniques will be motivated by specific codes which we will optimize step-by-step
- Techniques might seem problem-specific, but will (hopefully) demonstrate an overall strategy for optimizing kernels
- We will be using CPUs (e.g. not GPUs), but the strategies are generalizable

## Lecture section

- Introduction and refresh of notation, definitions, etc. (some overlap with "Performance Analysis" sessions)
- Finding a code's asymptotically achievable performance
- Understanding a code's current performance
- Steps to take to optimize, motivated by examples/exercises

# Outline

## Exercises

Three exercises to demonstrate some optimization strategies

1. Simple data-layout transformations

- "AoS to SoA" (*Array of Structures* to *Structure of Arrays*)

2. Optimizing matrix-matrix multiplications

- Blocking of data for mitigating cache misses

3. 3D laplacian-stencil code

- Multi-threading (OpenMP)
- Data layout transformations
- Blocking

# Hardware characteristics

## Peak floating point rate

- The theoretical, highest number of floating point operations that can be carried out by a computational unit
- Depends on: clock rate, vector length, FPUs per core, cores per socket

## Peak bandwidth

- The theoretical, highest number of bytes that can be read/written from/to some level of memory (L1,2,3 cache, RAM, etc.)
- For RAM: data rate, channels, ranks, banks

When optimizing, it is important to have these numbers in mind for the machine you're running on

# Julia

E.g. Julia compute nodes have the following characteristics:

- Peak FP ( $\gamma_{FP}$ )
  - 64-cores per node (1 KNL), clock: 1.3 GHz
  - Best case: two 512-bit fused multiply-and-adds per cycle (AVX-512)
- In double precision:  $2 \times (8 \text{ mul} + 8 \text{ add})$  per cycle = 32 flop/cycle
  - Therefore:  $1.3 \times 10^9 \text{ cycles/s} \times 32 \text{ flop/cycle} = 41.6 \text{ Gflop/s per core}$
  - 2662.4 Gflop/s per node
- Peak BW ( $\gamma_{IO}$ )
  - 115.2 GB/s to 96 GB DDR4 RAM assuming all channels populated
  - 400 GB/s to 16 GB MCDRAM
- Some (semi-)standard tools
  - On Linux, you can obtain processor details via `cat /proc/cpuinfo`.
  - You can obtain topology and memory info e.g. `hwloc`, `dmidecode` (latter requires access to `/dev/mem`)

# Julia

On Linux, you can obtain processor details via `cat /proc/cpuinfo`. For a Julia compute node:

```
processor      : 0
vendor_id     : GenuineIntel
...
model name    : Intel(R) Xeon Phi(TM) CPU 7230 @ 1.30GHz
...
cpu MHz       : 1261.914
cache size    : 1024 KB
physical id   : 0
siblings       : 256
core id        : 0
cpu cores     : 64
...
cpuid level   : 13
...
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 syscal...
```

# Computational kernels

## Sustained performance

- Sustained FP-rate: the measured, average number of floating point operations carried out by the kernel per unit time
  - `add`, `sub`, and `mul` count as 1 flop
  - `dev`, `sqrt`, `sin`, etc. count  $\geq 2$  flops. Depends on architecture
  - Count number of flops in kernel and divide by runtime
  - Alternatively, or for more complex codes, use performance counters

In our examples we will see cases of kernels where the flops are countable

- Sustained BW: the measured, average bytes read/written from main memory per unit time
  - As in the case of FP-rate, count bytes needed to be read and bytes needed to be written to and from RAM and divide by run time
  - **Maximum data reuse** assumption: once data is read from RAM, it never needs to be re-read

# An example

- Consider the following kernel operation:

$$Y_i^{ab} = A^{ac} \cdot X_i^{cb},$$

with:  $i = 0, \dots, L - 1, a, b, c = 0, \dots, N - 1, L \gg N$

- Naive implementation:

```
double Y[L][N][N];
double X[L][N][N];
double A[N][N];

/* ...
 * Initialize X[][][] and A[][]
 */
for(int i=0; i<L; i++) {
    for(int a=0; a<N; a++) {
        for(int b=0; b<N; b++) {
            Y[i][a][b] = 0;
            for(int c=0; c<N; c++) {
                Y[i][a][b] += A[a][c]*X[i][c][b];
            }
        }
    }
}
```

## Number of fp operations

- $N_{FP} = L \cdot 2 \cdot N^3$

## Number of bytes of I/O

- $N_{IO} = w \cdot (2 \cdot L \cdot N^2 + N^2)$
- $w$ : word-length in bytes, e.g.  $w = 4$  for single precision, 8 for double, etc.
- $2LN^2 \rightarrow O(L)$  if  $L \gg N$
- In any case, if  $N$  small enough,  $A$  should be kept in low-level cache

# An example

Note the assumption of data reuse

```
Y[i][0][0] = A[0][0]*X[i][0][0] + A[0][1]*X[i][1][0] + ... + A[0][N-1]*X[i][N-1][0];
Y[i][0][1] = A[0][0]*X[i][0][1] + A[0][1]*X[i][1][1] + ... + A[0][N-1]*X[i][N-1][1];
...
Y[i][1][0] = A[1][0]*X[i][0][0] + A[1][1]*X[i][1][0] + ... + A[1][N-1]*X[i][N-1][0];
Y[i][1][1] = A[1][0]*X[i][0][1] + A[1][1]*X[i][1][1] + ... + A[1][N-1]*X[i][N-1][1];
...
```

Elements of  $X$  and  $A$  are required multiple times. However we only count their loads once.

- Given some measurement of the run-time  $\bar{T}$

- FP-rate:  $\beta_{\text{FP}} = \frac{N_{\text{FP}}}{\bar{T}}$

- IO-rate:  $\beta_{\text{IO}} = \frac{N_{\text{IO}}}{\bar{T}}$

- This motivates defining an *intensity*  $I = \frac{N_{\text{FP}}}{N_{\text{IO}}}$

```
for(int i=0; i<L; i++) {
    for(int a=0; a<N; a++) {
        for(int b=0; b<N; b++) {
            Y[i][a][b] = 0;
            for(int c=0; c<N; c++) {
                Y[i][a][b] += A[a][c]*X[i][c][b];
            }
        }
    }
}
```

# Intensities

## Computational kernel intensity

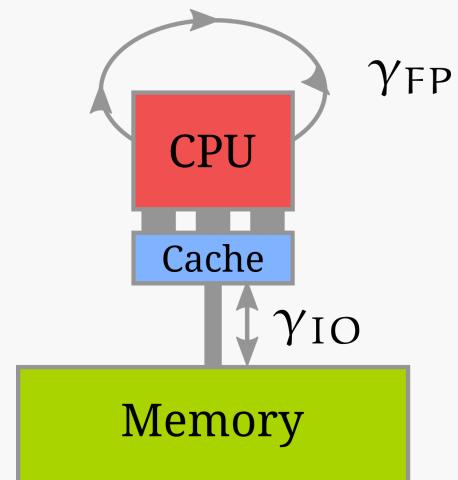
- Ratio of kernel floating point operations to bytes of I/O
- For our previous example:

- $I_k = \frac{N_{FP}}{N_{IO}} = \frac{2LN^3}{2wLN^2} = N/w$

- Note how the problem size  $L$  drops out  $\Rightarrow$  constant  $I_k$  irrespective of problem size
- E.g. for  $N = 3$  and double precision,  $I_k = 0.375$  flops/byte

## Machine flop/byte ratio

- Similarly to  $I_k$ , we can define the machine flop/byte ratio ( $I_m$ )
- $I_m = \frac{\gamma_{FP}}{\gamma_{IO}}$
- E.g. for a Julia (compute node):  $I_m \simeq 23 \frac{\text{flop}}{\text{byte}}$



# Intensities

Balance between kernel / hardware intensities

- $I_k \gg I_m$ : Kernel is "compute-bound" on this architecture. Higher  $\gamma_{FP}$  would lead to higher performance, but higher  $\gamma_{IO}$  would not.
- $I_k \ll I_m$ : Kernel is "bandwidth-" or "memory-bound" on this architecture. Higher  $\gamma_{IO}$  would lead to higher performance, but higher  $\gamma_{FP}$  would not.
- $I_k \simeq I_m$ : Kernel is balanced on this architecture. Ideal situation.

For the example kernel of the previous slides, on Julia

- $I_k \ll I_m \Rightarrow$  the kernel is memory-bound on Julia

Note the assumptions that enter  $I_k$  and  $I_m$

- $\gamma_{FP}$  considers all operations can be a sequence of multiply-and-add
- $\beta_{IO}$  assumes maximum data reuse
- $I_k$  constant if problem size  $L$  drops out

# Kernel computational intensity

Another example: matrix-matrix multiplication

- Consider a matrix-matrix multiplication:  $C_{M \times K} = A_{M \times N} \cdot B_{N \times K}$

```
double C[M][K];
double A[M][N];
double B[N][K];

for(int m=0; m<M; m++) {
    for(int k=0; k<K; k++) {
        C[m][k] = 0;
        for(int n=0; n<N; n++) {
            C[m][k] += A[m][n]*B[n][k];
        }
    }
}
```

- $N_{FP} = 2 \cdot M \cdot K \cdot N$
- $N_{IO} = w \cdot (M \cdot K + M \cdot N + N \cdot K)$
- $I_k = \frac{2}{w} \frac{1}{\frac{1}{M} + \frac{1}{N} + \frac{1}{K}}$
- E.g. for a square problem  $M = N = K$ ,  
 $\Rightarrow I_k = \frac{2}{3} \frac{N}{w}$

- This is an example of a kernel where  $I_k$  depends on the problem size.
- On a given architecture with  $I_m$ , the kernel transitions from bandwidth-bound to compute-bound as  $N$  increases.
- On Julia, the kernel is balanced when  $N \geq 12 \Rightarrow$  when the matrices are too large to fit in cache the problem is compute-bound.

# Optimization

Given a code you wish to optimize, for an architecture with  $I_m$

- What is  $N_{FP}$  and  $N_{IO}$  and what is the resulting  $I_k$ ?
- Is the kernel memory or compute bound on this architecture?
- What do you obtain for  $\beta_{FP}$  and  $\beta_{IO}$ 
  - For this one requires measuring the performance on the targeted architecture
- What are the ratios  $\frac{\beta_{FP}}{\gamma_{FP}}$  and  $\frac{\beta_{IO}}{\gamma_{IO}}$ ?

These are questions you need to answer before considering optimization

- After answering the above, we can start considering targeted optimizations for our kernel on the given machine
  - If your kernel is **memory-bound**, we should be trying to optimize for **memory I/O**. Ideally we try to achieve a  $\frac{\beta_{IO}}{\gamma_{IO}} \rightarrow 1$ .
  - If your kernel is **compute-bound**, we should be trying to optimize for a higher **FP-rate**. Ideally we try to achieve a  $\frac{\beta_{FP}}{\gamma_{FP}} \rightarrow 1$ .
- In practice, however, we will see how these two strategies in general are combined to optimize both memory- and compute-bound kernels.

# Optimization

On modern systems, the compiler and the architecture will do a lot for you

- Loop unrolling, prefetching, out-of-order execution, auto-vectorization, etc.

We will motivate our optimizations by considering specific examples where a significant speedup can be achieved on top of `-O3`

## Vectorization

- We will see how to use *vector intrinsic functions* which compile to SIMD instructions
- At first sight, this optimization seems to target compute bound kernels
- However the data transformations required to efficiently vectorize a compute kernel also prove beneficial for the I/O efficiency

```
float ar = creal(a);
float ai = cimag(a);
float aux0[] = { ar, ar, ar, ar };
float aux1[] = {-ai, ai, -ai, ai};
register __m128 va_re = __mm_load_ps(aux0);
register __m128 va_im = __mm_load_ps(aux1);
register __m128 x0, x1, y0;
...

```

```
for(int i=0; i<L; i+=2) {
    y0 = __mm_load_ps((float *)&y[i]);
    x0 = __mm_load_ps((float *)&x[i]);
    x1 = __mm_shuffle_ps(x0, x0, MASK);
    y0 = __mm_fmadd_ps(va_im, x1, y0);
    y0 = __mm_fmadd_ps(va_re, x0, y0);
    __mm_store_ps((float *)&y[i], y0);
}
```

# Optimization

On modern systems, the compiler and the architecture will do a lot for you

- Loop unrolling, prefetching, out-of-order execution, auto-vectorization, etc.

We will motivate our optimizations by considering specific examples where a significant speedup can be achieved on top of `-O3`

## Data layout transformations

- Transformations for mitigation of cache misses (improve temporal and spatial cache locality)
- Transformations which can assist vectorization or auto-vectorization

## Cache awareness

- Similar to the above, this optimization will allow us to modify code parameters according to cache characteristics (*blocking* or *tiling*)

# Optimization

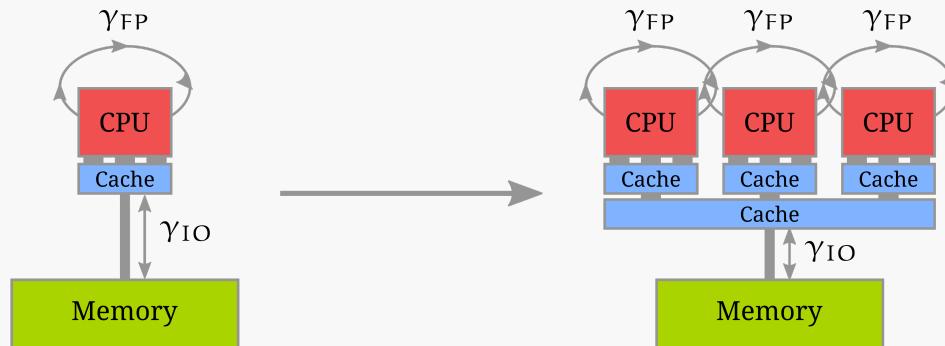
On modern systems, the compiler and the architecture will do a lot for you

- Loop unrolling, prefetching, out-of-order execution, auto-vectorization, etc.

We will motivate our optimizations by considering specific examples where a significant speedup can be achieved on top of `-O3`

## Multi-threading

- Some simple `OpenMP` can go a long way



- For compute-bound kernels, this effectively multiplies  $\gamma_{FP}$
- For memory-bound kernels, allows better saturation of  $\gamma_{IO}$

# Vectorization

Most modern processors have some vectorization capabilities

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
    y[i] = a*x[i] + y[i];
```

# Vectorization

Most modern processors have some vectorization capabilities

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)
    y[i] = a*x[i] + y[i];
```

```
for(int i=0; i<L; i+=4) {
    y[i] = a*x[i] + y[i];
    y[i+1] = a*x[i+1] + y[i+1];
    y[i+2] = a*x[i+2] + y[i+2];
    y[i+3] = a*x[i+3] + y[i+3];
}
```

# Vectorization

Most modern processors have some vectorization capabilities

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)  
    y[i] = a*x[i] + y[i];
```

```
float4 va = {a,a,a,a};  
for(int i=0; i<L; i+=4) {  
    float4 vy;  
    float4 vx;  
    load4(vx, &x[i]);  
    load4(vy, &y[i]);  
    vy = va*vx + vy;  
    store4(&y[i], vy);  
}
```

```
for(int i=0; i<L; i+=4) {  
    y[i] = a*x[i] + y[i];  
    y[i+1] = a*x[i+1] + y[i+1];  
    y[i+2] = a*x[i+2] + y[i+2];  
    y[i+3] = a*x[i+3] + y[i+3];  
}
```

**Note:** the above is pseudo-code, namely `load4`,  
`float4`, etc. are simplifications.

# Vectorization

Most modern processors have some vectorization capabilities

- SSE, AVX, AltiVec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)  
    y[i] = a*x[i] + y[i];
```

```
float4 va = {a,a,a,a};  
for(int i=0; i<L; i+=4) {  
    float4 vy;  
    float4 vx;  
    load4(vx, &x[i]);  
    load4(vy, &y[i]);  
    vy = va*vx + vy;  
    store4(&y[i], vy);  
}
```

```
for(int i=0; i<L; i+=4) {  
    y[i] = a*x[i] + y[i];  
    y[i+1] = a*x[i+1] + y[i+1];  
    y[i+2] = a*x[i+2] + y[i+2];  
    y[i+3] = a*x[i+3] + y[i+3];  
}
```

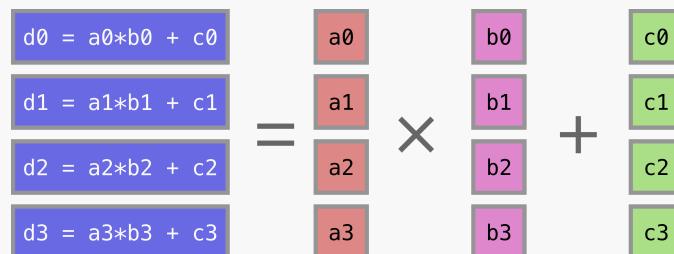
**Note:** the above is pseudo-code, namely `load4`, `float4`, etc. are simplifications.

- In many cases, the compiler will generate vector instructions (auto-vectorization)
- If not, we will learn here how to explicitly program using these vector instructions, via *intrinsic functions*

# Vectorization

## Intrinsics

- These are extensions to C and architecture-specific
- Here we will see some Intel intrinsics, though most of what we will learn is generalizable to other architectures. In the best case, they differ in the function naming convention
- For an exhaustive reference on Intel intrinsics, the best resource is at:  
<https://software.intel.com/sites/landingpage/IntrinsicsGuide>
  - Lists intrinsics for all versions of SSE to AVX and KNL
  - Note the header requirements depending on the extension version under which the function you would like to use is defined (e.g. `<xmmmintrin.h>`, `<immintrin.h>`, etc.)



```
d = _mm256_fmaddd_pd(a, b, c)
```

# Vectorization

## Intrinsics - short intro

- Vector types defined in `<?mmmintrin.h>`:
  - `_m128`: 128-bit single precision ⇒ can pack four
  - `_m128d`: 128-bit double precision ⇒ can pack two
  - `_m256` (`_m256d`): 256-bit float (double), packs eight (four)
- `_mm_load_pd()`: "load packed double", e.g.:

```
double *x;
posix_memalign(&x, 32, L*sizeof(double));
double __attribute__((aligned(32))) a;
_m128d vx, va;
_mm_load_pd(vx, &x[i]);
_mm_load_pd(va, &a);
```

- **Alignment restrictions:** `&x[i]` must be a multiple of the vector length.
- Use `posix_memalign()` to allocate
- Use `__attribute__((aligned(32)))` for static
- `_mm_load_ps()`, `_mm256_load_ps()`, etc.

# Vectorization

## Intrinsics - short intro

- Store:
  - `_mm_store_ps()`, `_mm256_store_ps()`, `_mm_store_pd()`, etc.
  - Same restriction on alignment as in the case of load
- Arithmetic:
  - `c = _mm_add_pd(a, b);`,  $c = a + b$  with  $a, b, c$  of type `_m128d`
  - `c = _mm256_mul_ps(a, b);`,  $c = a \times b$  with  $a, b, c$  of type `_m256`
  - In general, `_mm_` prepends 128-bit vector instructions (SSE) and `_mm256_` prepends 256-bit instructions (AVX)
  - If processor supports fused-multiply-add (`fma` flag in `/proc/cpuinfo`), then:  
`d = _mm256_fmadd_pd(a, b, c);`,  $d = a \times b + c$  with  $a, b, c, d$  of type `_m256d`

# Vectorization

## Intrinsics - short intro

- Shuffles, swizzles,
  - Instructions which allow permuting the elements within a vector
- One classical demonstration is when doing complex multiplications, e.g.:  
$$z_n = x_n \cdot y_n, \text{ with } x, y, z \text{ complex, e.g: } x_n = x_n^r + i x_n^i \Rightarrow$$
$$z_n^r = x_n^r \cdot y_n^r - x_n^i \cdot y_n^i$$
$$z_n^i = x_n^r \cdot y_n^i + x_n^i \cdot y_n^r$$
- A complex type is typically implemented as an array of two elements:

```
for(int n=0; n<N; n++) {  
    z[n][0] = x[n][0]*y[n][0] - x[n][1]*y[n][1]; /* z[n][0]: real part of z[n] */  
    z[n][1] = x[n][1]*y[n][0] + x[n][0]*y[n][1]; /* z[n][1]: imag part of z[n] */  
}
```

# Vectorization

## Intrinsics - short intro

```
for(int n=0; n<N; n++) {
    z[n][0] = x[n][0]*y[n][0] - x[n][1]*y[n][1]; /* z[n][0]: real part of z[n] */
    z[n][1] = x[n][1]*y[n][0] + x[n][0]*y[n][1]; /* z[n][1]: imag part of z[n] */
}
```

- One (naive) way to implement this in intrinsics:

```
_m256d vx0, vx1, vy0, vy1, aux0, aux1;
_m256d mp = {-1.0, +1.0};
for(int n=0; n<N; n++) {
    _mm_load_pd(vx0, &x[n][0]);
    _mm_load_pd(vy0, &y[n][0]);
    vx1 = _mm_shuffle_pd(vx0, vx0, 0b10); /* swap real and imag of vx0 and put into vx1 */
    vy1 = _mm_shuffle_pd(vy0, vy0, 0b11); /* put the imag of vy0 into both elems of vy1 */
    vy0 = _mm_shuffle_pd(vy0, vy0, 0b00); /* put the real of vy0 into both elems of vy0 */
    vx1 = _mm_mul_pd(mp, vx1);           /* flip sign of 1st elem of vx1 and store in vx1 */
    aux0 = _mm_mul_pd(vx0, vy0);
    aux1 = _mm_mul_pd(vx1, vy1);
    vz = _mm_add_pd(aux0, aux1);
    _mm_store_pd(&z[n][0], vz);
}
```

# Vectorization

## Intrinsics - short intro

- General form of shuffle: `c = _mm_shuffle_pd(a, b, mask)`
- 

```
mask = 0b00 → c = {a[0], b[0]}    mask = 0b01 → c = {a[1], b[0]}
mask = 0b10 → c = {a[0], b[1]}    mask = 0b11 → c = {a[1], b[1]}
```

---

- In general, rightmost bits of `mask` select elements of `a` and leftmost bits select elements of `b`
- A better implementation may be (SSE3):

```
for(int n=0; n<N; n++) {
    z[n][0] = x[n][0]*y[n][0];
    z[n][0] -= x[n][1]*y[n][1];
    z[n][1] = x[n][1]*y[n][0];
    z[n][1] += x[n][0]*y[n][1];
}
```

```
c = hadd(a, b) →
c[0] = a[0]+a[1],
c[1] = b[0]+b[1]
```

```
m128d mp = {+1.0, -1.0};
for(int n=0; n<N; n++) {
    _mm_load_pd(vx0, &x[n][0]);
    _mm_load_pd(vy0, &y[n][0]);
    vx1 = _mm_shuffle_pd(vx0, vx0, 0b10); /* swap re-im */
    vx0 = _mm_mul_pd(mp, vx0);
    aux0 = _mm_mul_pd(vx0, vy0);
    aux1 = _mm_mul_pd(vx1, vy0);
    vz = _mm_hadd_pd(aux0, aux1);
    _mm_store_pd(&z[n][0], vz);
}
```

# Data layout

## Data layout transformations

- Better cache locality
  - The data are ordered in a way as close to the order in which they will be accessed in your kernels as possible
  - The data are reordered such that when an element needs to be accessed multiple times, these multiple accesses are issued very close to each other
  - Optimizes for temporal and for spatial locality
- Easier vectorization
  - The data are ordered such that the same operation can be applied to consecutive elements
    - Assists the compiler in detecting auto-vectorization opportunities
    - Assists the programmer in implementing the intrinsics

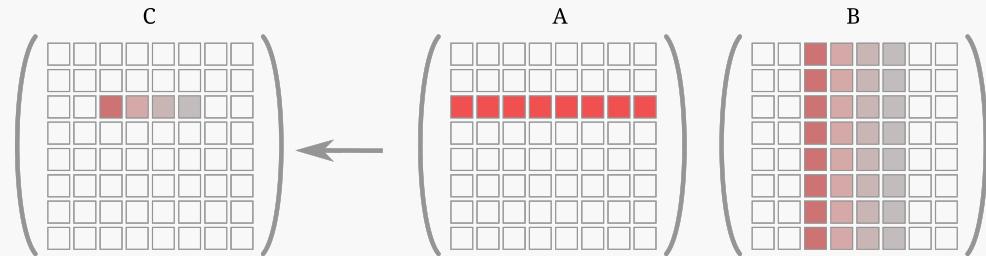
# Data layout

## 1. Matrix-matrix multiplication

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}$$

$A_{M \times N}, B_{N \times M}, C_{M \times M}$

```
for(int i=0; i<M; i++) {  
    for(int j=0; j<M; j++) {  
        C[i*M + j] = 0;  
        for(int k=0; k<N; k++) {  
            C[i*M + j] += A[i*N + k]*B[k*M + j];  
        }  
    }  
}
```



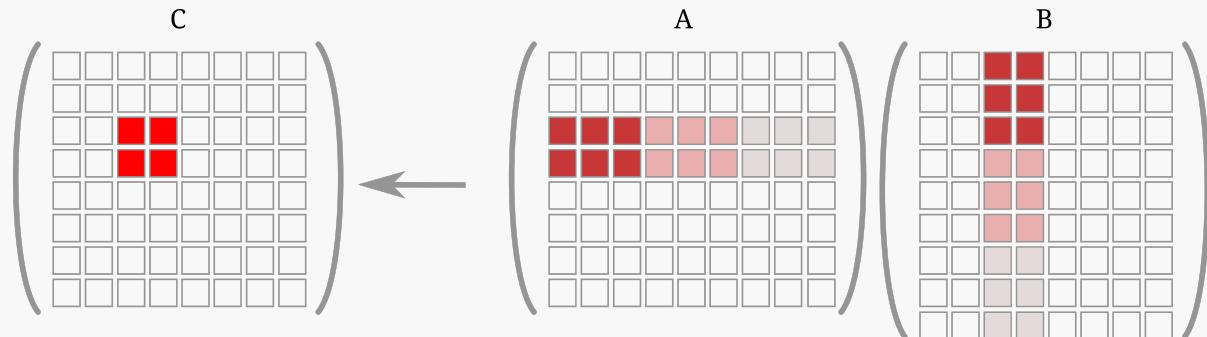
- Repeated reading of columns of  $B$
- If  $N$  too large, a whole row of  $A$  may not fit into L2 cache

# Data layout

## 1. Matrix-matrix multiplication

Consider a blocked version

```
for(int i=0; i<M; i+=BM)
    for(int j=0; j<M; j+=BM) {
        Cb [:BM] [:BM] = 0;
        for(int k=0; k<N; k++) {
            Ab [:BM] [:BN] = A[i:i+BM] [k:k+BN];
            Bb [:BN] [:BM] = B[k:k+BN] [j:j+BM];
            for(int ib=0; ib<BM; ib++)
                for(int jb=0; jb<BM; jb++)
                    for(int kb=0; kb<BN; kb++)
                        Cb[ib][jb] += Ab[ib][kb] * Bb[kb][jb];
            C[i:i+BM] [j:j+BM] = Cb [:BM] [:BM];
        }
    }
```

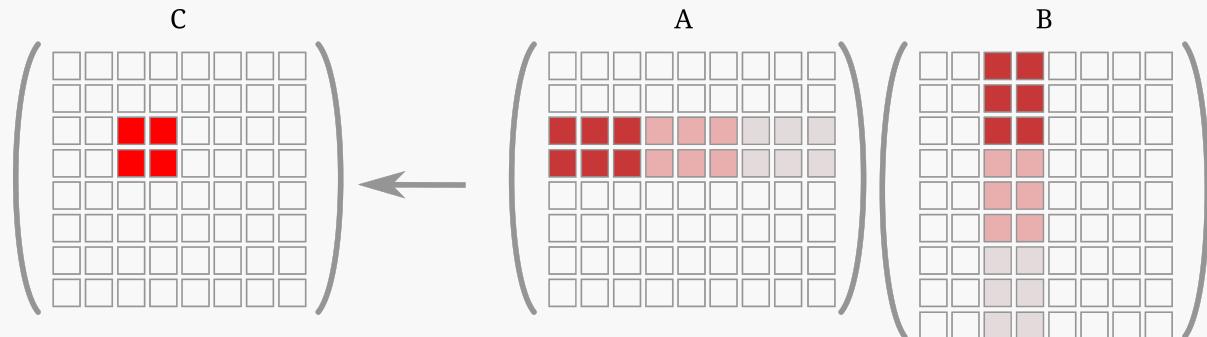


# Data layout

## 1. Matrix-matrix multiplication

Consider a blocked version

```
for(int i=0; i<M; i+=BM)
    for(int j=0; j<M; j+=BM) {
        Cb[:BM] [:BM] = 0;
        for(int k=0; k<N; k++) {
            Ab[:BM] [:BN] = A[i:i+BM] [k:k+BN];
            Bb[:BM] [:BN] = B[j:j+BM] [k:k+BN]; /* Transpose B! */
            for(int ib=0; ib<BM; ib++)
                for(int jb=0; jb<BM; jb++)
                    for(int kb=0; kb<BN; kb++)
                        Cb[ib][jb] += Ab[ib][kb] * Bb[jb][kb];
            C[i:i+BM] [j:j+BM] = Cb[:BM] [:BM];
        }
    }
```



# Data layout

## 2. AoS to SoA

### AoS: Array of structures

Arrays of structures arise naturally when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {  
    float x;  
    float y;  
    float z;  
} coords;
```

You can now allocate an array of `coords`:

```
size_t L = 1000;  
coords *arr = malloc(sizeof(coords)*L);
```

Now assume you would like to get an array of the distances of the coordinates from the origin:

```
for(int i=0; i<L; i++)  
    r[i] = sqrt(arr[i].x*arr[i].x +  
                arr[i].y*arr[i].y +  
                arr[i].z*arr[i].z);
```

You can see how the choice of data layout makes auto-vectorization difficult. A common way around this is to use a Structure of Arrays (SoA) rather than an Array of Structures.

# Data layout

## 2. AoS to SoA

Define a structure of arrays, this is similar to how one programs for GPUs:

```
typedef struct {  
    float *x;  
    float *y;  
    float *z;  
} coords;
```

And now allocate each element of `coords` separately:

```
coords arr;  
arr.x = malloc(sizeof(float)*L);  
arr.y = malloc(sizeof(float)*L);  
arr.z = malloc(sizeof(float)*L);
```

You can see how it is more obvious vectorize the distance calculation after unrolling:

```
for(int i=0; i<L; i+=4) {  
    r[i] = sqrt(arr.x[i]*arr.x[i]+arr.y[i]*arr.y[i]+arr.z[i]*arr.z[i]);  
    r[i+1] = sqrt(arr.x[i+1]*arr.x[i+1]+arr.y[i+1]*arr.y[i+1]+arr.z[i+1]*arr.z[i+1]);  
    r[i+2] = sqrt(arr.x[i+2]*arr.x[i+2]+arr.y[i+2]*arr.y[i+2]+arr.z[i+2]*arr.z[i+2]);  
    r[i+3] = sqrt(arr.x[i+3]*arr.x[i+3]+arr.y[i+3]*arr.y[i+3]+arr.z[i+3]*arr.z[i+3]);  
}
```

# Data layout

## 3. Stencil codes

Another application of the data re-ordering is in stencil codes. Consider a 2-D stencils operation:

$$\phi_{x,y} \leftarrow 4\psi_{x,y} - (\psi_{x+1,y} + \psi_{x-1,y} + \psi_{x,y+1} + \psi_{x,y-1})$$

```
for(int y=0; y<L; y++)  
for(int x=0; x<L; x++) {  
    B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
}
```

Consider an unrolled implementation of this, with `x` running fastest and `y` slowest:

```
for(int y=0; y<L; y++)  
for(int x=0; x<L; x+=4) {  
    B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
    B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x] + A[y+1][x+1] + A[y-1][x+1]);  
    B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);  
    B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);  
}
```

If  $A[y][x:x+4]$  is fit into a vector register, you can see how many shuffles of the elements are required to vectorize this operations.

# Data layout

## 3. Stencil codes

```
for(int y=0; y<L; y++)  
for(int x=0; x<L; x+=4) {  
    B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
    B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x] + A[y+1][x+1] + A[y-1][x+1]);  
    B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);  
    B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);  
}
```

An alternative is to change the data layout so that we vectorize over distant elements.  
I.e. we can reshape the data layout:

from:  $A[y][x] \leftarrow A + y*L + x$

to:  $vA[y0][x][y1] = A[y0+y1*L/4][x]$ , and re-write the kernel:

```
for(int y=0; y<L/4; y++)  
for(int x=0; x<L; x++) {  
    vB[y][x][0] = 4*vA[y][x][0] - (vA[y][x+1][0] + vA[y][x-1][0] + vA[y+1][x][0] + vA[y-1][x][0]);  
    vB[y][x][1] = 4*vA[y][x][1] - (vA[y][x+1][1] + vA[y][x-1][1] + vA[y+1][x][1] + vA[y-1][x][1]);  
    vB[y][x][2] = 4*vA[y][x][2] - (vA[y][x+1][2] + vA[y][x-1][2] + vA[y+1][x][2] + vA[y-1][x][2]);  
    vB[y][x][3] = 4*vA[y][x][3] - (vA[y][x+1][3] + vA[y][x-1][3] + vA[y+1][x][3] + vA[y-1][x][3]);  
}
```

# Data layout

## 3. Stencil codes

Shuffling of the elements is still required but restricted to the boundaries. E.g. for periodic boundary conditions, the original code at the  $y=0$  boundary would be:

```
y = 0;
for(int x=0; x<L; x++) {
    B[0][x] = 4*A[0][x] - (A[0][x+1] + A[0][x-1] + A[1][x] + A[L-1][x]);
}
```

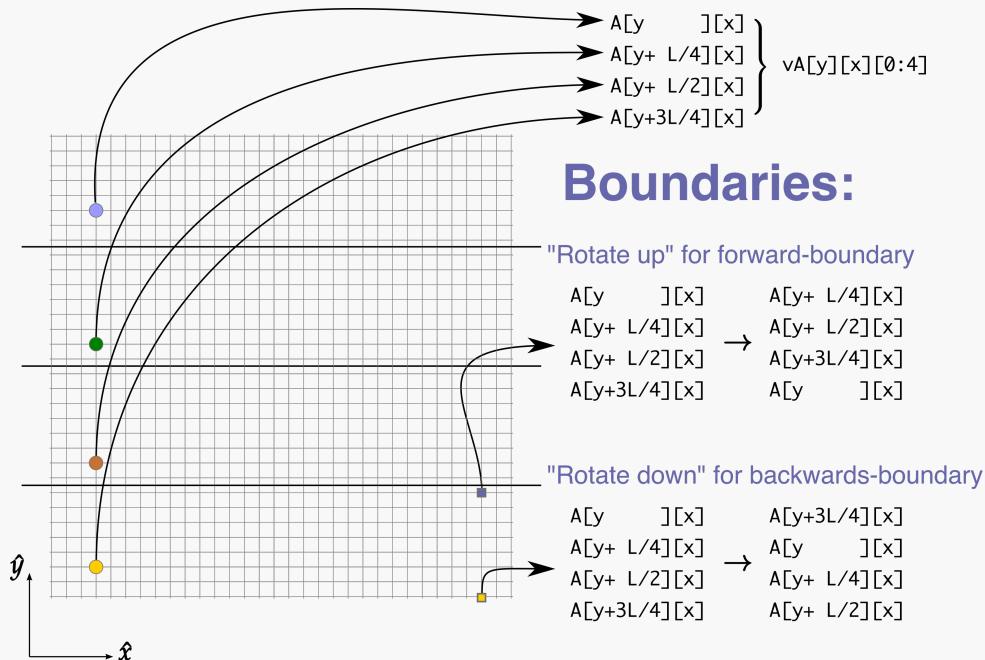
which requires shuffling of the boundary element at  $y=L/4-1$  of  $vA$ :

```
y = 0;
for(int x=0; x<L; x++) {
    vB[0][x][0] = 4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/4-1][x][3]);
    vB[0][x][1] = 4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/4-1][x][2]);
    vB[0][x][2] = 4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/4-1][x][1]);
    vB[0][x][3] = 4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/4-1][x][0]);
}
```

# Data layout

## 3. Stencil codes

```
y = 0;  
for(int x=0; x<L; x++) {  
    vB[0][x][0] = 4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/4-1][x][3]);  
    vB[0][x][1] = 4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/4-1][x][2]);  
    vB[0][x][2] = 4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/4-1][x][1]);  
    vB[0][x][3] = 4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/4-1][x][0]);  
}
```



# Exercises

```
git clone -n https://github.com/g-koutsou/CDS-1.git  
cd CDS-1  
git checkout trainee
```

The exercises are prepared for: `julia.fz-juelich.de`

For compiling codes, use Intel's C compiler: `module load openmpi`

For plotting, use Python 3: `module load python/3.6.1 matplotlib/2.1.0`

# AoS to SoA

In the first exercise we will look into two optimizations:

1. Going from an Array-of-Structures to a Structure-of-Arrays,
2. which will assist the (auto-)vectorization of our kernel

The kernel does the following: assuming an array of 4-vectors:  $a_i = (ct_i, x_i, y_i, z_i)$  compute the squared "norm":  $s_i = (ct_i)^2 - (x_i^2 + y_i^2 + z_i^2)$

Navigate to: [CDS-1/0ptim/Ex/AoS-to-SoA/](#).

# AoS to SoA

In the first exercise we will look into two optimizations:

1. Going from an Array-of-Structures to a Structure-of-Arrays,
2. which will assist the (auto-)vectorization of our kernel

The kernel does the following: assuming an array of 4-vectors:  $a_i = (ct_i, x_i, y_i, z_i)$  compute the squared "norm":  $s_i = (ct_i)^2 - (x_i^2 + y_i^2 + z_i^2)$

Navigate to: [CDS-1/0ptim/Ex/AoS-to-SoA/](#).

```
while(1) {
    double t0 = stop_watch(0);
    for(int i=0; i<NREP; i++)
        comp_s(arr, L);
    t0 = stop_watch(t0)/(double)NREP;
    t0acc += t0;
    t1acc += t0*t0;
    if(n > 2) {
        double ave = t0acc/n;
        double err = sqrt(t1acc/n -
                           ave*ave)/sqrt(n);
        if(err/ave < 0.1) {
            t0acc = ave;
            t1acc = err;
            break;
        }
    }
    n++;
}
```

`stop_watch()` is just a system timer:

```
double
stop_watch(double t0) {
    struct timeval tp;
    gettimeofday(&tp, NULL);
    double t1 = tp.tv_sec + tp.tv_usec*1e-6;
    return t1-t0;
}
```

- Call the kernel `NREP` times
- Do this repeatedly. Stop when variation in time to call kernel `NREP` times is smaller than 10%

# AoS to SoA

In the first exercise we will look into two optimizations:

1. Going from an Array-of-Structures to a Structure-of-Arrays,
2. which will assist the (auto-)vectorization of our kernel

The kernel does the following: assuming an array of 4-vectors:  $a_i = (ct_i, x_i, y_i, z_i)$  compute the squared "norm":  $s_i = (ct_i)^2 - (x_i^2 + y_i^2 + z_i^2)$

Navigate to: [CDS-1/0ptim/Ex/AoS-to-SoA/](#).

`stop_watch()` is just a system timer:

```
double  
stop_watch(double t0) {  
    struct timeval tp;  
    gettimeofday(&tp, NULL);  
    double t1 = tp.tv_sec + tp.tv_usec*1e-6;  
    return t1-t0;  
}
```

`x`, `y`, `z`, `t`, and `s` are elements of a `struct`

```
typedef struct {  
    float x, y, z, t;  
    float s;  
} st_coords;
```

The computation of `s` is a loop over an array of `st_coords`:

```
void  
comp_s(st_coords *arr, int L)  
{  
#pragma omp parallel for  
    for(int i=0; i<L; i++) {  
        float x = arr[i].x;  
        float y = arr[i].y;  
        float z = arr[i].z;  
        float t = arr[i].t;  
        float s = t*t - (x*x + y*y + z*z);  
        arr[i].s = s;  
    }  
    return;  
}
```

**Note** that we are using OpenMP and parallelizing over 128 threads

# AoS to SoA

For the first task you will modify `space-time-aos.c`. You have instructions tagged with `_TODO_A_`.

After correcting for `beta_fp` and `beta_io`, you need to compile:

```
module load openmpi  
make A
```

and submit the prepared job script:

```
sbatch sub-A.job
```

```
void  
comp_s(st_coords *arr, int L)  
{  
#pragma omp parallel for  
    for(int i=0; i<L; i++) {  
        float x = arr[i].x;  
        float y = arr[i].y;  
        float z = arr[i].z;  
        float t = arr[i].t;  
        float s = t*t - (x*x + y*y + z*z);  
        arr[i].s = s;  
    }  
    return;  
}  
...  
double beta_fp = 0 /* _TODO_A_: insert number of */  
/* Gflop/sec based on timing */;  
double beta_io = 0 /* _TODO_A_: insert number of */  
/* Gbyte/sec based on timing */;
```

This will run the code for various lengths `L`. The output is in `aos.txt`. There is a Python script so that you can visualize the output. You will need to:

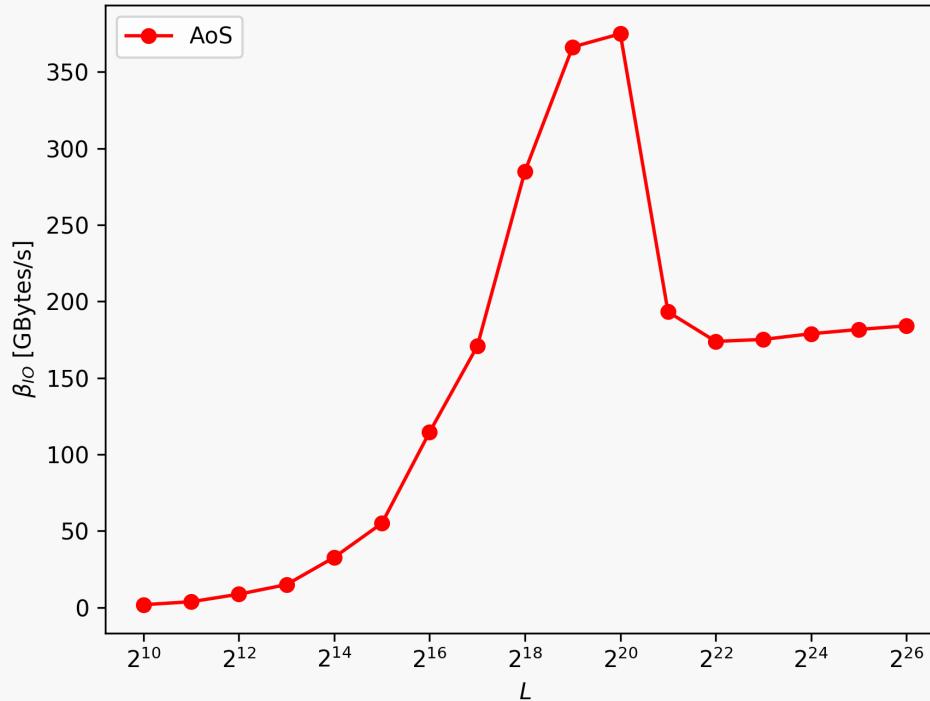
```
module load python/3.6.1 matplotlib/2.1.0,
```

run: `python3 plot-A.py` and then this will produce `A.png`.

# AoS to SoA

## Task A

The resulting plot should look like:



# AoS to SoA

In Task B you need to modify `space-time-soa.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

`space-time-soa.c` implements a structure of arrays of length `VL` rather than an array of structures:

```
typedef struct {
    float x[VL];
    float y[VL];
    float z[VL];
    float t[VL];
    float s[VL];
} st_coords;
```

Once done, as in Task A, make and submit:

```
make B
sbatch sub-B.job
```

and plot the result using: `python3 plot-B.py` to produce `B.png`.

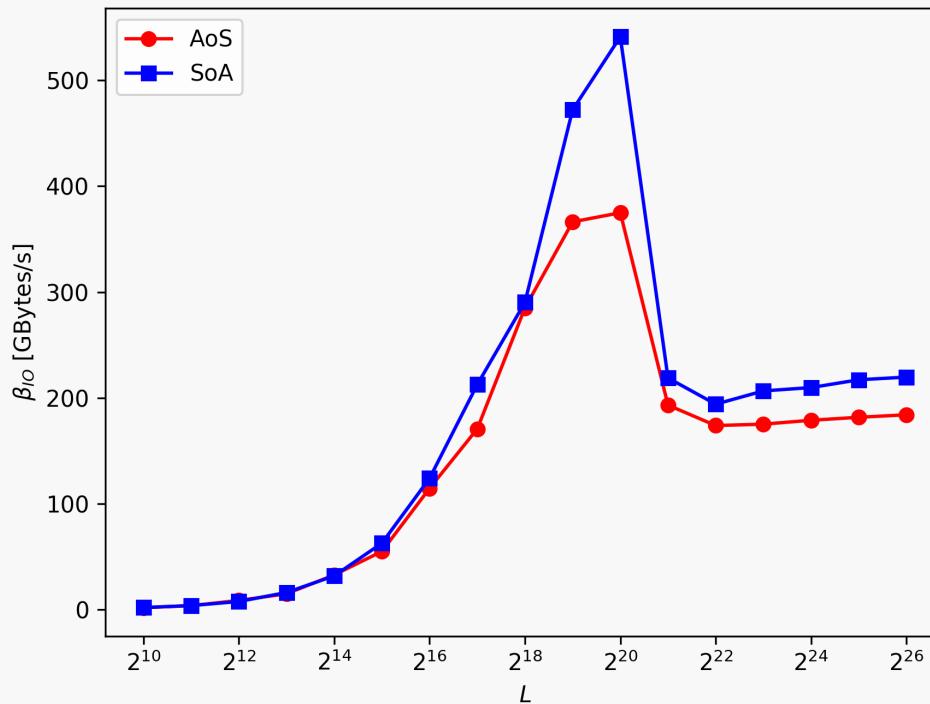
For the pieces tagged with `_TODO_B_` you need to implement `comp_s()` to operate on this new data layout:

```
/*
 * compute: s = t**2 - x**2 - y**2 - z**2,
 * with s, t, x, y, z member variables of
 * length VL of the st_coords structure
 */
void
comp_s(st_coords *arr, int L)
{
    /* _TODO_B_: complete the kernel */
    #pragma omp parallel for
    ...
```

# AoS to SoA

## Task B

The resulting plot should look like:



# AoS to SoA

## Task C

In Task C, you will use the same program, `space-time-soa.c`. But you will run it multiple times, varying `VL`.

You are not required to modify code. Look at `sub-C.job`:

```
#!/bin/bash
#SBATCH -J soa
#SBATCH -N 1
#SBATCH -o soa-%j.out
#SBATCH -e soa-%j.err
module load openmpi

export OMP_NUM_THREADS=128

for VL in 2 4 8 16 32 64 128 256; do
    make -B VL=${VL} B
    for L in 1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 8388608 167772
        srun numactl --membind 1 ./space-time-soa $L
    done | tee soa-vl${VL}.txt
done
```

This will compile and run the program for all values of `VL` in the list. Submit with:

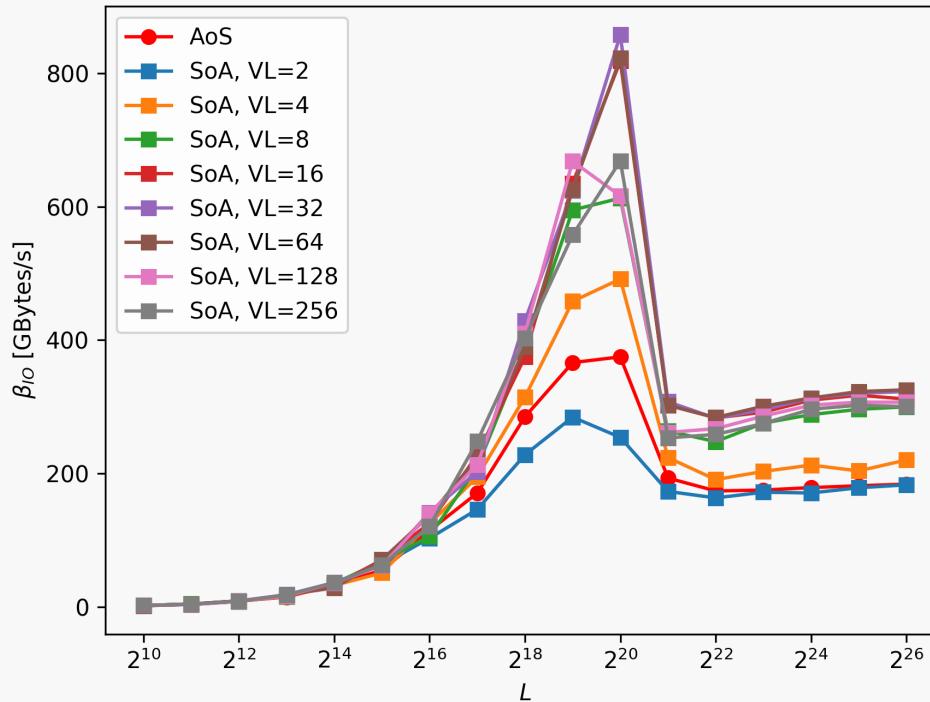
```
sbatch sub-C.job
```

and run: `python3 plot-C.py`. This will produce `C.png`.

# AoS to SoA

## Task C

The resulting plot should look like:



For which value of VL do we achieve best performance? Why?

# AoS to SoA

In Task D you need to modify `space-time-soa-v.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

`space-time-soa-v.c` implements the same structure of arrays as in `space-time-soa.c`

For the pieces tagged with `_TODO_D_` you need to implement `comp_s()` in intrinsics. Once done, as in the previous tasks, make and submit:

```
make D  
sbatch sub-D.job
```

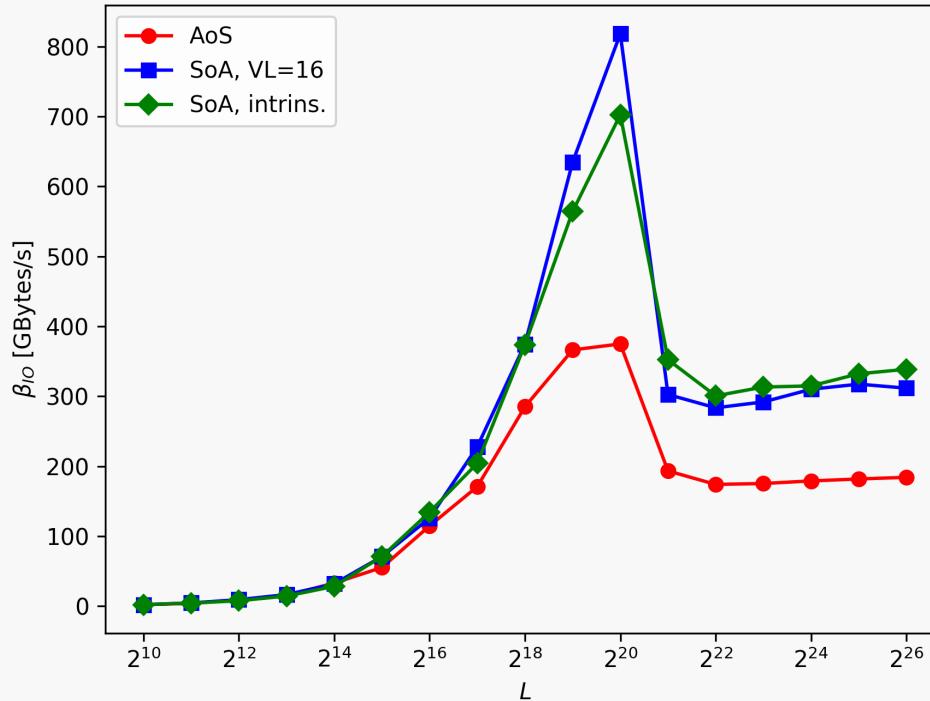
and plot: `python3 plot-D.py`.

```
void  
comp_s(st_coords *arr, int L)  
{  
#pragma omp parallel for  
for(int i=0; i<L/VL; i++) {  
    register __m512 x = _mm512_load_ps(&arr[i].x[0]);  
    register __m512 y = _mm512_load_ps(&arr[i].y[0]);  
    register __m512 z = _mm512_load_ps(&arr[i].z[0]);  
    register __m512 t = _mm512_load_ps(&arr[i].t[0]);  
    register __m512 s0;  
    /*  
     _TODO_D_  
  
     Complete this function using  
     intrinsics so that it computes:  
  
     arr[i].s[:] = arr[i].t[:]**2 - (  
         arr[i].x[:]**2 +  
         arr[i].y[:]**2 +  
         arr[i].z[:]**2  
     )  
  
     You will use:  
  
     _mm512_mul_ps();  
     _mm512_fmadd_ps();  
     _mm512_fmsub_ps();  
  
     and the temporary variable s0.  
    */  
  
    /*  
     Store s0 to arr[i].s[:]  
    */  
    _mm512_stream_ps(&arr[i].s[0], s0);  
}  
  
return;  
}
```

# AoS to SoA

## Task D

The resulting plot should look like:



# MatMul

This exercise deals with a matrix-matrix multiplication

Navigate to `CDS-1/0ptim/Ex/MatMul/`. There is only one file `mm.c`.

In the first task (task A), you will need to complete the `_TODO_A_` tags. Here this requires:

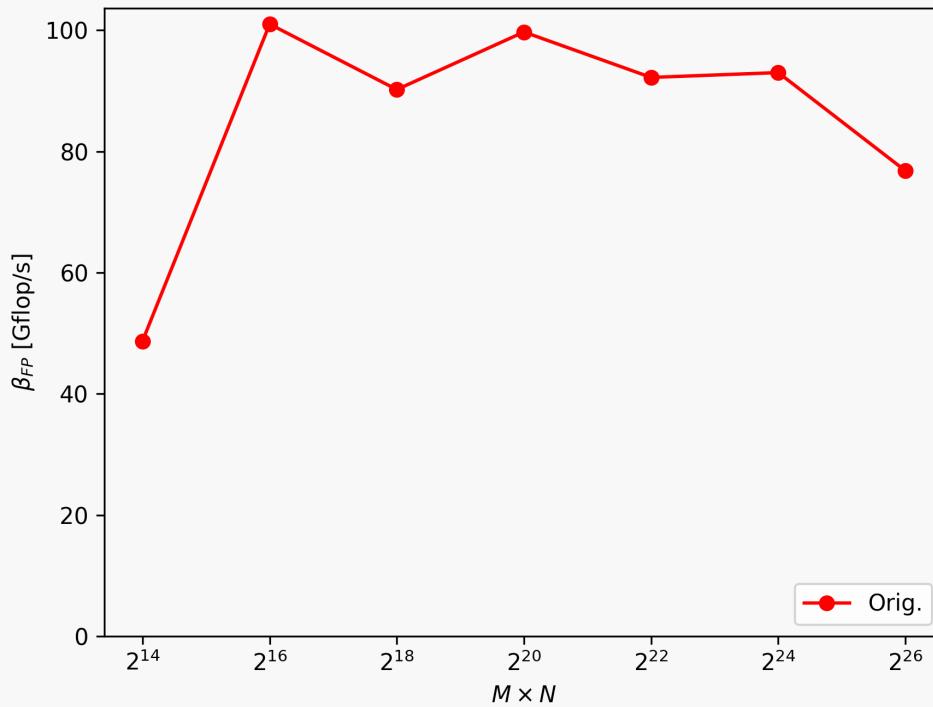
1. Computing `N_fp`, so that the flops are reported correctly.
2. Implement naively the matrix matrix multiplication in `mat_mul()`.

```
void
mat_mul(double *C, int M, int N, double *A, double *B)
{
#pragma omp parallel for collapse(2)
    for(int i=0; i<M; i++)
        for(int j=0; j<N; j++)
            C[i*M + j] = 0;

#pragma omp parallel for collapse(2)
    for(int i=0; i<M; i++)
        for(int k=0; k<N; k++)
            for(int j=0; j<N; j++)
/*
    _TODO_A_
    Implement the matrix-matrix multiplication naively
*/
    ...
}
```

# MatMul

When done: `make A`, `sbatch sub-A.` job, and then you can plot the result with `python3 plot-A.py`. The job script runs the `mm-orig` program for multiple values of `M` and `N` with `M == N`. The result should look like:



# MatMul

In Task B you will implement a blocked version of the matrix-matrix multiplication.  
Find the tags with `_TODO_B_`.

```
void
mat_mul_blocked(double *C, int M, int N, double *A, double *B)
{
    double Ab[BM*BN];
    double Bb[BM*BN];
    double Cb[BM*BM];
    for(int i=0; i<M; i+=BM)
        for(int j=0; j<N; j+=BM) {

            /* Set Cb to zero */
            for(int ib=0; ib<BM; ib++)
                for(int jb=0; jb<BN; jb++) {
                    Cb[ib*BN + jb] = 0;
                }

            /*
             * _TODO_B_
             *
             * Implement the blocked matrix-matrix multiplication
             * Steps:
             *
             * 0. Loop over k-blocks
             * 1. Fill Ab with the block that begins from A[i:][k:]
             * 2. Fill Bb with the block that begins from B[k:][j:]
             * 3. Do the little matrix matrix multiplication Cb = Ab*Bb
             *
             */
            /*
             * Copy Cb to C[i][j];
             */
            ...
        }
}
```

# MatMul

While developing the matrix-matrix multiplication, you can make `B` and run for a small `M, N`, e.g.:

```
./mm-blck 64 64
```

The code checks that the two versions (blocked and original) produce the same result and will print to the screen if there is a discrepancy, e.g.:

```
ORIG: M = 128, N = 128, took: 5.25e-04 sec, P = 7.99e+03 Mflop/s
BLCK: M = 128, N = 128, took: 4.79e-04 sec, P = 8.76e+03 Mflop/s, BM = 16, BN = 16
Non zero diff: 6.559491e-02
```

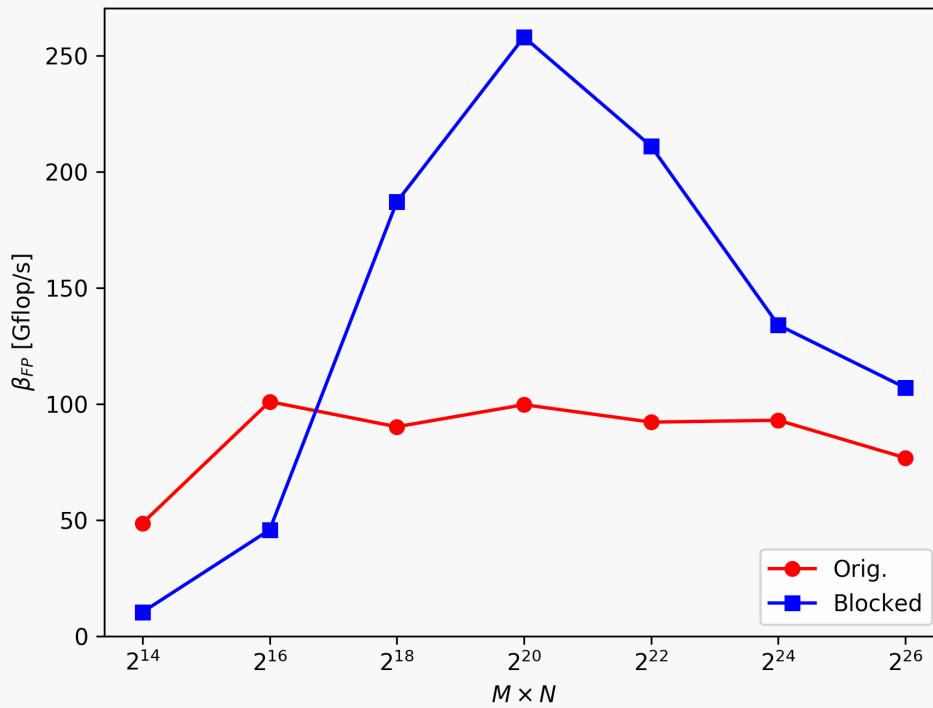
When done, and the two versions agree, submit with: `sbatch sub-B.job`, and after the job completes you can plot the result with `python3 plot-B.py`. Note that the block lengths are defined at compile time. They are defined at the beginning:

```
/*
 * Block length in M direction
 */
#ifndef BM
#define BM 64
#endif
```

```
/*
 * Block length in N direction
 */
#ifndef BN
#define BN 64
#endif
```

# MatMul

The resulting plot should look like:



# 3D Laplacian

- Consider the 3D "gauge" Laplacian.

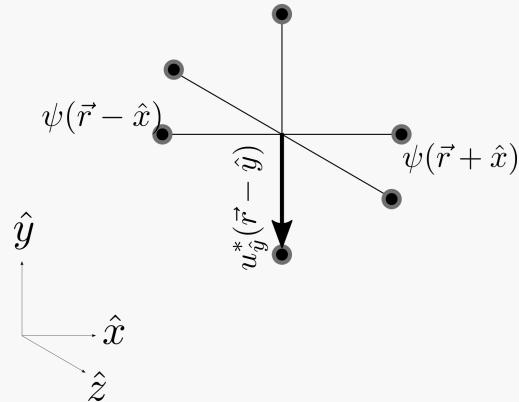
$$\phi(\vec{r}) = \nabla \psi(\vec{r}) = 6\psi(\vec{r}) - \sum_{i=1}^{N_d} [u_i^\wedge(\vec{r})\psi(\vec{r} + \hat{i}) + u_i^*(\vec{r} - \hat{i})\psi(\vec{r} - \hat{i})]$$

with  $N_d = 3$  the number of dimensions,

$u_i^\wedge(\vec{r}) \in \mathbb{C}$ , and  $|u_i^\wedge(\vec{r})| = 1$

$\psi(\vec{r}) \in \mathbb{C}$

- $2 \cdot N_d$  "u"s at each site  $\vec{r}$ , but only  $N_d$  unique "u"s
- Conjugation flips the direction of  $u$ , i.e.  $u_{-\hat{i}}^\wedge(\vec{r}) = u_i^*(\vec{r})$



- 
- $\hat{i}$  unit vector in direction  $i$ ,  $(u_0^\wedge(\vec{r}), u_1^\wedge(\vec{r}), u_2^\wedge(\vec{r}))$ , or  $(u_z^\wedge(\vec{r}), u_y^\wedge(\vec{r}), u_x^\wedge(\vec{r}))$

# 3D Laplacian

- Conjugate gradient (CG) to solve

$$\nabla x = b$$

initial guess  $x$

$$r_0 = b - \nabla x$$

$$p = r_0$$

loop until  $\|r_0\|$  small enough

$$\alpha = \frac{r_0^\dagger r_0}{p^\dagger \nabla p}$$

$$x = \alpha p + x$$

$$r_1 = -\alpha \nabla p + r_0$$

$$\beta = \frac{r_1^\dagger r_1}{r_0^\dagger r_0}$$

$$p = \beta p + r_1$$

$$r_0 = r_1$$

Linear algebra already implemented  
in exercise: `xdotx()`, `xdoy()`, `axpy()`,  
`aypx()`, `xmy()`

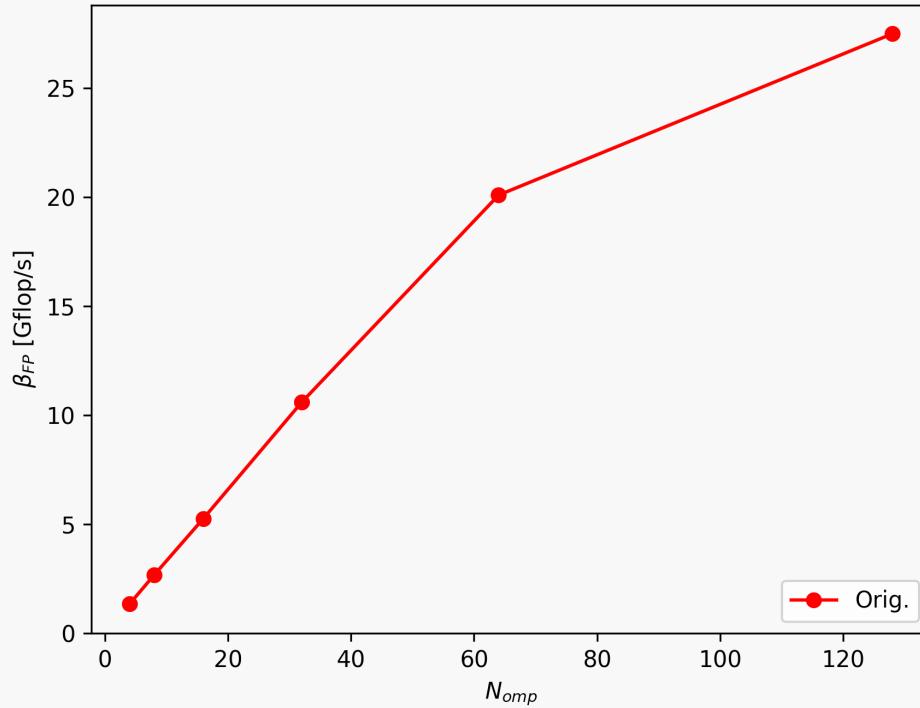
```
$ srun numactl --membind 1 ./lapl 128
 0, res = +1.000000e+00
 1, res = +1.669268e-01
 2, res = +3.343321e-02
 3, res = +6.682492e-03
 4, res = +1.339953e-03
 5, res = +2.682193e-04
 6, res = +5.365824e-05
 7, res = +1.079675e-05
 8, res = +2.181284e-06
 9, res = +4.423248e-07
10, res = +9.009035e-08
11, res = +1.844173e-08
12, res = +3.792212e-09
13, res = +7.814874e-10
14, res = +1.617658e-10
15, res = +3.366356e-11
16, res = +7.020740e-12
17, res = +1.470992e-12
18, res = +3.094643e-13
19, res = +6.524311e-14
20, res = +1.382112e-14
21, res = +2.943272e-15
22, res = +6.290599e-16
23, res = +1.345215e-16
24, res = +2.883775e-17
25, res = +6.205892e-18
26, res = +1.337031e-18
27, res = +2.886712e-19
```

Converged after 27 iterations, res = +3.781296e-14  
Time in lapl(): +7.828e-02 sec/call, 1.34e+00 Gflop/s, 1.0

# 3D Laplacian

- Navigate to CDS-1/Optim/Ex/Lapl-3D/
- [lapl.c](#) implements a CG algorithm which solves  $\nabla x = b$  for a random  $b$  and gauge  $u$
- The [\\_T0D0\\_A](#) pieces need completion
- This includes the bulk of the Laplacian operation in function `lapl()` at [lapl.c:145](#)
- After you're done compile with `make A` and submit the job `sub-A.job`. You can then plot with `python plot-A.py`

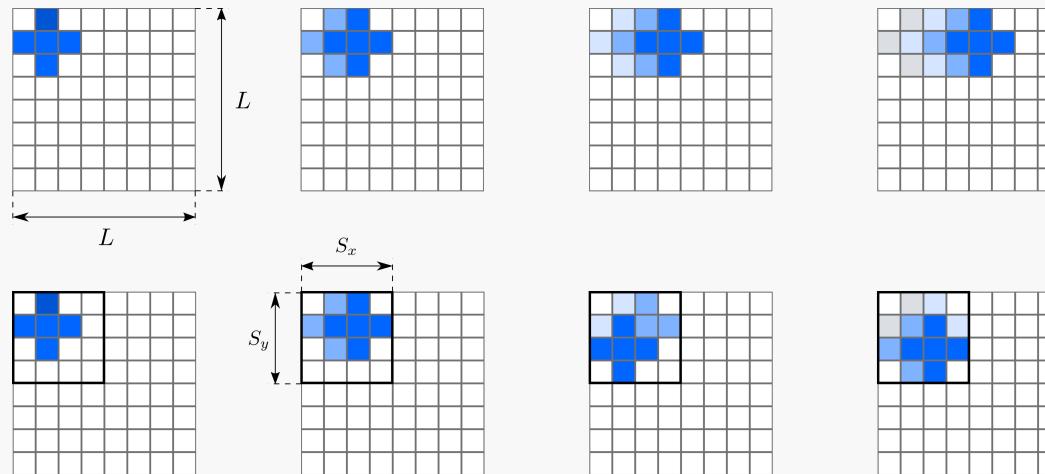
# 3D Laplacian



The plot is the performance vs the number of OMP threads

# 3D Laplacian

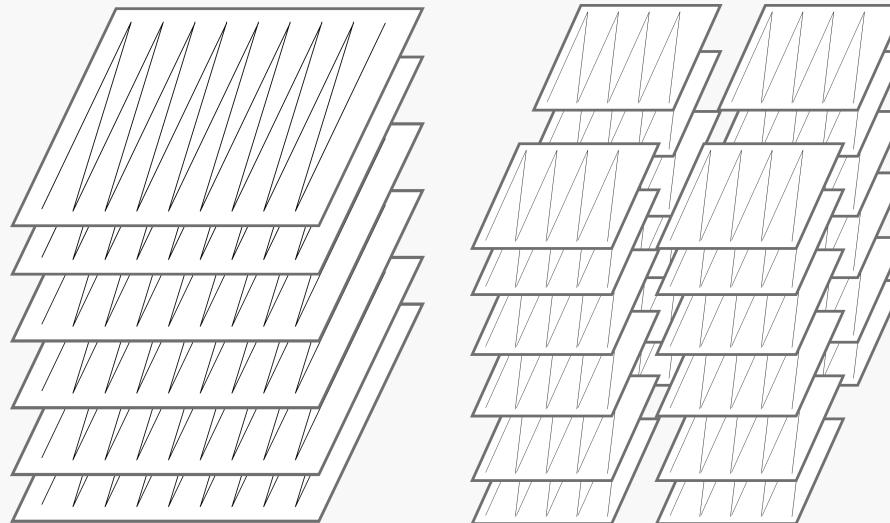
- In Task B, you need to modify `laplt.c` ("t" stands for "tiled")
- This is to implement a "blocked" (or "tiled") version of the stencil operation in `lapl()`



- Block in  $x$  and  $y$  directions
- Distribute OMP threads so that each thread processes a block
- To check correctness, you should obtain the same CG iterations as the original version

# 3D Laplacian

- In Task B, you need to modify `laplt.c` ("t" stands for "tiled")
- This is to implement a "blocked" (or "tiled") version of the stencil operation in `lapl()`



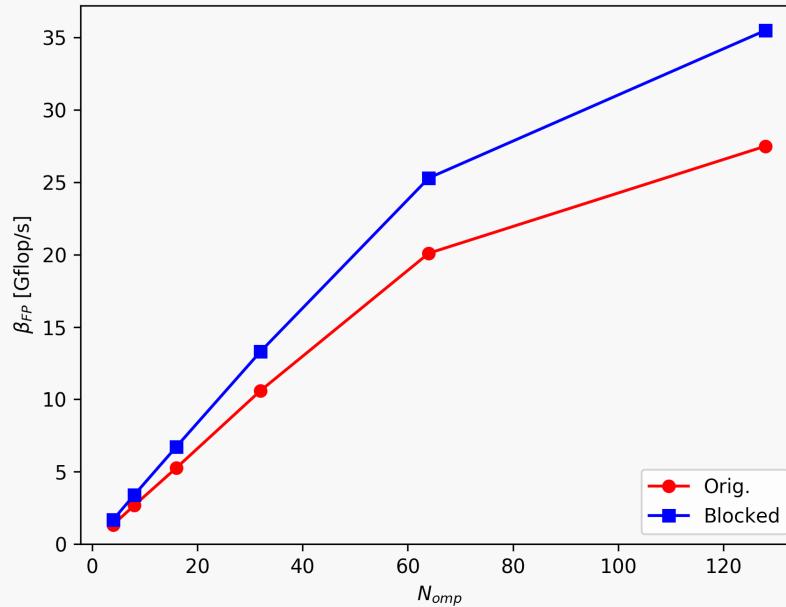
- $A[z][y][x] \rightarrow bA[x_0][y_0][z][y_1][x_1] = A[z][y_1 + y_0*S_y][x_1 + x_0*S_x]$
- Distribute OMP threads so that each thread processes a block
- To check correctness, you should obtain the same CG iterations as the original version

# 3D Laplacian

- The `_TODO_B_` tag the pieces you need to modify in `laplt.c`. This includes:
  - Completing the bulk calculation of `lapl()`, now that the lattices are blocked  
[laplt.c:284](#)
- Handling of the boundaries is taken care of [laplt.c:316](#)
- Compile with `make B`, submit with `sbatch sub-B.job`, and plot with `python3 plot-B.py`

# 3D Laplacian

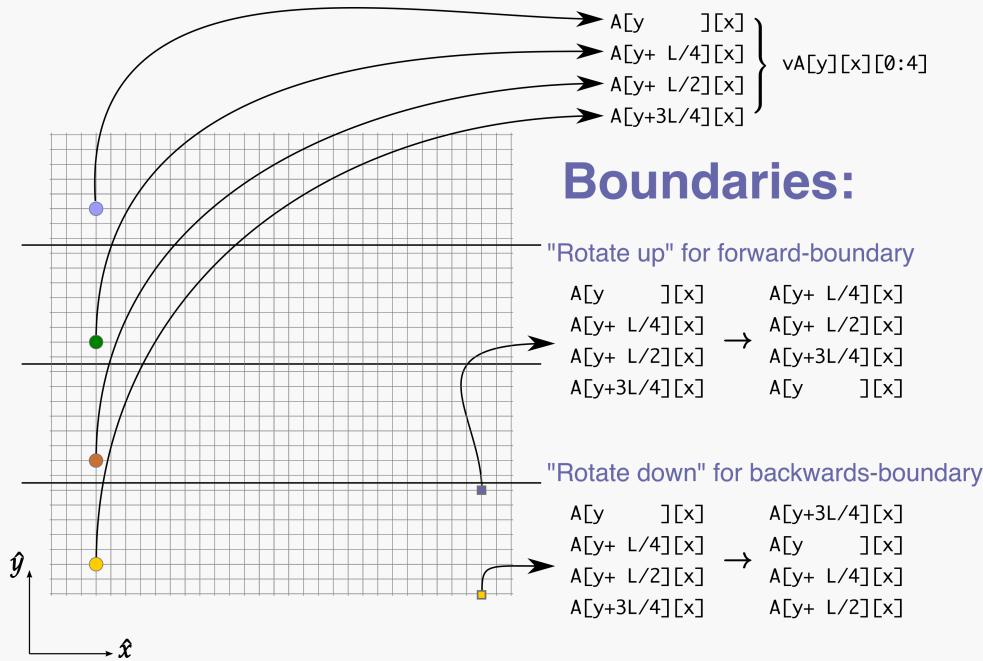
The result of Task B should look like:



Performance of the blocked code compared to the original version

# 3D Laplacian

- Task C involves implementing a version of `lapl()` which is easier for vectorization



- $A[z][y][x][\text{reim}] \rightarrow$   
 $vA[z_0][y][x][\text{reim}][z_1] = A[z_0 + z_1*L/4][y][x][\text{reim}]$
- Notice shift required on boundaries

# 3D Laplacian

- Task C involves implementing a version of `lapl()` which is easier for vectorization
- $A[z][y][x][\text{reim}] \rightarrow$   
 $vA[z0][y][x][\text{reim}][z1] = A[z0 + z1*L/4][y][x][\text{reim}]$

E.g. for the case of 2D:

```
for(int y=0; y<L/4; y++)  
for(int x=0; x<L; x++) {  
    vB[y][x][0] = 4*vA[y][x][0] - (vA[y][x+1][0] + vA[y][x-1][0] + vA[y+1][x][0] + vA[y-1][x][0]);  
    vB[y][x][1] = 4*vA[y][x][1] - (vA[y][x+1][1] + vA[y][x-1][1] + vA[y+1][x][1] + vA[y-1][x][1]);  
    vB[y][x][2] = 4*vA[y][x][2] - (vA[y][x+1][2] + vA[y][x-1][2] + vA[y+1][x][2] + vA[y-1][x][2]);  
    vB[y][x][3] = 4*vA[y][x][3] - (vA[y][x+1][3] + vA[y][x-1][3] + vA[y+1][x][3] + vA[y-1][x][3]);  
}
```

- Notice shift required on boundaries, e.g. for the  $y = 0$  boundary in the 2D problem:

```
y = 0;  
for(int x=0; x<L; x++) {  
    vB[0][x][0] = 4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/4-1][x][3]);  
    vB[0][x][1] = 4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/4-1][x][2]);  
    vB[0][x][2] = 4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/4-1][x][1]);  
    vB[0][x][3] = 4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/4-1][x][0]);  
}
```

# 3D Laplacian

- The code to be modified is now `laplv.c`, the parts missing in tagged with `_TODO_C_`
- Modify `laplv.c`, compile with `make C`, run with `sbatch sub-C.job` and plot using `python3 plot-C.py`.
- After completing these parts you should obtain similar CG history as before
- Here the bulk calculation of the Laplacian is implemented. Also the `OpenMP` pragma has been inserted
- You need to complete the pieces that do the shuffling for the boundaries at [laplv.c:254](#)

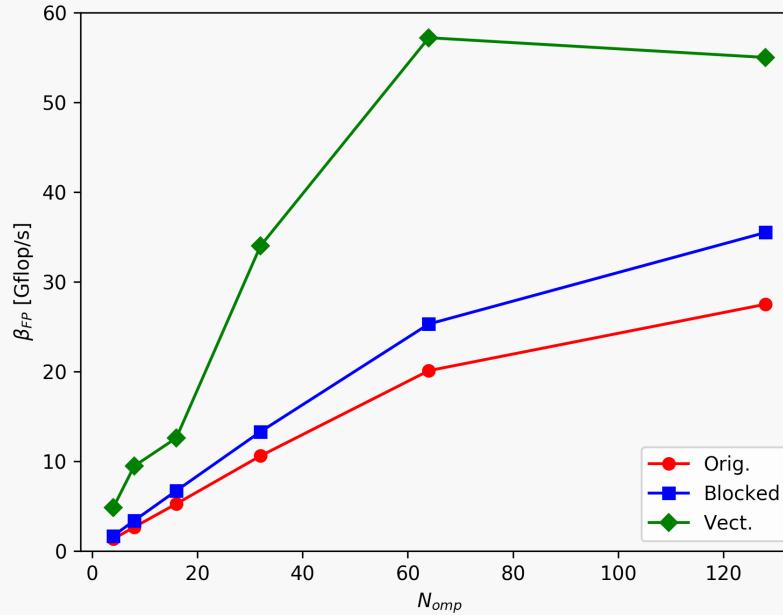
```
if(z == 0) {  
/* _TODO_C_  
*  
* Use shift_dn() or shift_up() to appropriately  
* shift the necessary structures, when on the  
* boundary  
*/  
}
```

```
if(z == lz-1) {  
/* _TODO_C_  
*  
* Use shift_dn() or shift_up() to appropriately  
* shift the necessary structures, when on the  
* boundary  
*/  
}
```

- With `shift_up()` and `shift_dn()` are already defined for you

# 3D Laplacian

Result of Task C should look like:



Vector-friendly version compared to original and blocked versions