Seminar for Applied Mathematics, ETHZ

20 Oct. 2017

HPCSE/Y2017/unit4
Basics of data independence:

\[
\text{for}(i=m; \ i<n; \ i++) \ x[i] = f(x[i-k]);
\]

If order of execution follows C rules, \( x[i-k] \) is computed before \( x[i] \), \( k > 0 \). It is clear that the maximum number that may be computed in parallel is

\[
\text{no. i's computed in parallel} \leq k
\]

For example, \( k = 2 \):

\[
\begin{align*}
  x[m] &= f(x[m-2]); \\
  x[m+1] &= f(x[m-1]); \\
  x[m+2] &= f(x[m]); \\
  x[m+3] &= f(x[m+1]); \\
  x[m+4] &= f(x[m+2]); \\
  \vdots & \quad \vdots
\end{align*}
\]
Case \( k \geq 0 \):

\[
\text{for}(i=0; i<n-m; i++) \ x[i] = f(x[i+k]);
\]

The number \( i \)'s that may be computed in parallel is arbitrary. For example, \( k = 2 \):

\[
\begin{align*}
x[0] &= f(x[2]) ;
x[1] &= f(x[3]) ;
x[2] &= f(x[4]) ;
x[3] &= f(x[5]) ;
x[4] &= f(x[6]) ;
\end{align*}
\]

Notice that no stored value of \( x[i] \) is ever read again within the loop. For example, \( x[2] \) is set but not used afterwards in this loop.
However, in the C ordering rules (use -fvectorize gcc switch).

```c
#pragma GCC ivdep
for(i=0;i<n;i++){
    x[i] = f(x[i+k]);
}
```

for $k > 0$, $n - m$ (all of the loop) can be computed in parallel:

\[
S_1 \leftarrow [x_{i+k}, x_{i+k+1}, x_{i+k+2}, ...] \\
S_2 \leftarrow f(S_1) \text{ vector of results} \\
[x_i, x_{i+1}, ...] \leftarrow S_2
\]

because although the segments $[x_{i+k}, x_{i+k+1}, ...]$ and $[x_i, x_{i+1}, ...]$ may overlap, $S_1$ has a copy of the old data, so no $x_i$ is ever written onto before it is copied, then used.
Vectors, or segments, warps $S_r$, are purely symbolic and may be one of

- $S_r$ is a ”vector” of registers (Cray, Intel SSE/AVX)
- $S_r$ is a collection of threads. NVIDIA uses warps of threads: These can run up to 32/time.

Basic procedure is **unrolling loops**, done by you or compiler.

```plaintext
for(i=0;i<n;i++) { y[i] = f(x[i]); }
```

can be expanded, e.g. by 4’s. Let $p = n \mod 4$:

```plaintext
y[0] = f(x[0]); ... y[p-1] = f(x[p-1]); // p<4
for(i=p;i<n;i+=4){
    y[i] = f(x[i]); y[i+1] = f(x[i+1]);
    y[i+2] = f(x[i+2]); y[i+3] = f(x[i+3]);
}
```
These are done in groups (vectors) of 4 at a time. Or \( m \) at a time. **Why would a compiler do this?**

All functional units are pipelined. If, for example, we want to multiply an array of **integers** \( \vec{A} \) times \( \vec{B} \) to get array \( \vec{C} \): for \( i = 0... \), \( C_i = A_i \times B_i \), with bytes 3,2,1,0.

\[
A_i = [A_{i3}, A_{i2}, A_{i1}, A_{i0}]
\]
\[
B_i = [B_{i3}, B_{i2}, B_{i1}, B_{i0}]
\]
\[
C_i = [C_{i3}, C_{i2}, C_{i1}, C_{i0}]
\]

\( C_{ij} = A_{ij} \times B_{ij} + \) carry from \( A_{i,j-1} \times B_{i,j-1} \)

when stage \( j \) is done with \( A_{i,j} \times B_{i,j} \), it can be used for \( A_{i+1,j} \times B_{i+1,j} \), until reaching the last \( i \). In this case, after 4 cycles the pipeline is full, then we get 1 result/clock-cycle.
Two forms of vectorization: long $SL$, and short $SL$.

$$SL = \text{vector length, no. of indep. data}$$

$$PL = \text{pipeline length. This is the latency.}$$

$$\text{speed-up} = \text{scalar/vector timing ratio}$$

Scalar mode takes $PL \cdot SL$ cycles, but in parallel only $SL + PL$, so

$$\text{speed-up} = \frac{PL \cdot SL}{SL + PL}$$

When $SL$ large compared to $PL$, (Cray, Fujitsu, NEC, Hitachi)

$$\text{speedup} \approx PL$$

But if $PL$ large (memory pipe dominates: Pentiums, NVIDIA),

$$\text{speedup} \approx SL$$
Figure: L-stage multiply pipeline: $\vec{C} = \vec{A} \ast \vec{B}$
Figure: $\vec{C} = \vec{A} \ast \vec{B}$ pipeline.

Another example: carry-save adder.
GPUs strategy is **latency hiding**. $i = 1 : SL$ in ea. $SL$ segment.

**Figure:** Alignment of operations to fill in latencies. We assume $SL|n$, while loop variable $i$ is stepped by $SL$. 

\[
\begin{align*}
S_0 & \leftarrow A_i \\
& \quad \text{wait} \\
S_1 & \leftarrow f_1(S_0) \\
S_2 & \leftarrow f_2(S_1) \\
B_i & \leftarrow S_2 \\
& \quad \vdots \\
\end{align*}
\]

\[
\begin{align*}
S_4 & \leftarrow A_{i+SL} \\
& \quad \text{wait} \\
S_5 & \leftarrow f_1(S_4) \\
S_6 & \leftarrow f_2(S_5) \\
B_{i+SL} & \leftarrow S_6 \\
& \quad \vdots \\
\end{align*}
\]

\[
\begin{align*}
S_0 & \leftarrow A_i \\
S_4 & \leftarrow A_{i+SL} \\
& \quad \text{wait} \\
S_1 & \leftarrow f_1(S_0) \\
S_5 & \leftarrow f_1(S_4) \\
S_2 & \leftarrow f_2(S_1) \\
S_3 & \leftarrow f_2(S_5) \\
B_i & \leftarrow S_2 \\
B_{i+SL} & \leftarrow S_6 \\
& \quad \vdots \\
\end{align*}
\]
Memory basics

Principle of Data Locality: The safest assumption about the next data to be used is that they are the same or nearby the last used.

What are the basic components of memory?

- **SRAM** = *static random access memory* is typically CMOS (complementary metal oxide semiconductor) and is logically constructed from flip-flops. SRAM is fast and has no need to be refreshed, although all data are lost if the power fails.

- **DRAM** = *dynamic random access memory* in its simplest form contains one transistor and one capacitor. Capacitors leak, thus DRAM must be refreshed every few milliseconds. Some variants now clock at 715 MHz.
Figure: Memory speed vs. CPU
**Figure:** Left: 6 transistor SRAM cell. Right: 4 transistor SRAM cell.

**Figure:** Hyundai SRAM chip.
Figure: Left: DRAM array (MOSFET). Right: MT4C1024 DRAM 1 megabit or $2^{20}$ (128 kB).
Figure: A generic virtual memory machine with cache.
Figure: Elements of a simple cache.
Figure: Mapping of virtual addresses to real addresses.
Figure: Component fields of a virtual address.
It is also important to know that instruction execution is pipelined. On Intel architectures, instructions are **wide format**.

- long instruction formats are often strings of $\mu$-ops, which can be re-ordered.
- operations can be overlapped - so operations, too, are pipelined.
- in fact, modern hardware has **branch prediction** which keeps a history and re-orders as needed.

```c
if(e(x)){
    y = f(x);
} else {
    y = g(x);
}
```
Figure: Instruction execution unit for Pentium IV.
Figure: Storage unit with TLB (left) and cache (right): information flow has $X$ connected to $X$, $Y$ connected to $Y$. 
Figure: Associative cache designs.
Considerations which effect SIMD algorithms? Example sdot: vectors on Intel I-5 and I-7. Single precision (32-bit) vector registers $S_k$ are 4 words/each: **these vectors must be aligned on cache line boundaries**.

![Alignment of SSE/Altivec data.](image)

**Figure:** Alignment of SSE/Altivec data.
Inner product:

\[ \text{dot} = (x, y) = \sum_{i=1}^{n} x_i y_i. \]

For simplicity, let \( n = q \cdot SL \) (i.e. \( n = 4q \)): a partial reduction \( i = 1, \ldots, SL \),

\[
(S_7)_i = \text{partial}_i = \sum_{k=0}^{q-1} x_{i+k} \cdot SL \cdot y_{i+k+SL},
\]

\[
\begin{align*}
S_7 & \leftarrow [0, 0, 0, \ldots] & \quad & \text{// initialize } S_7 \text{ to zero} \\
& \qquad \text{// loop over segments: } m += SL \text{ each time} \\
S_0 & \leftarrow [x_m, x_{m+1}, x_{m+2}, \ldots] & \quad & \text{// read } SL \text{ of } x \\
S_1 & \leftarrow [y_m, y_{m+1}, y_{m+2}, \ldots] & \quad & \text{// read } SL \text{ of } y \\
S_3 & \leftarrow S_1 \ast S_2 & \quad & \text{// segment of } x \ast y \\
S_7 & \leftarrow S_7 + S_3 & \quad & \text{// accumulate partial result}
\end{align*}
\]
Do you see the difficulty?

Assume $x, y$ are aligned and $n = 4 \cdot q$. Switches for G-4 altivec:

```
gcc -O3 -faltivec sdottest.c -lm
```
float sdot0(int n, float *x, float *y)
{
    // Altivec version of sdot. W. Petersen 2002
    float *xp,*yp,psum[4],sum=0.0;
    int i,nsegs; /* nsegs = q */
    vector float S0,S1,S7=(vector float)(0.0,0.0,0.0,0.0);
    xp = x; yp = y;
    S0 = vec_ld(0,xp); xp += 4; /* load x */
    S1 = vec_ld(0,yp); yp += 4; /* load y */
    nsegs = (n >> 2) - 1;
    for(i=0;i<nsegs;i++){
        S7 = vec_madd(S0,S1,S7); /* part sum of 4 */
        S0 = vec_ld(0,xp); xp += 4; /* load next 4 x */
        S1 = vec_ld(0,yp); yp += 4; /* load next 4 y */
    }
    S7 = vec_madd(S0,S1,S7); vec_st(S7,0,psum);
    for(i=0;i<4;i++) sum += psum[i];
    return(sum);
}
#include "xmmmintrin.h"

void saxpy_sse(int n, float a, float *x, float *y) {
    int i; // SSE version of SAXPY, Roman Geus 03/2001
    sse_t a_vec;
    for (i = 0; i < 4; i++)
        a_vec.sf[i] = a; // vector with 4 a vals
    movups_m2r(a_vec, XMM0);
    n = n/4;
    for (i = 0; i < n; i++) {
        movups_m2r(*(((sse_t *) x)+i), XMM1); // next 4 x's
        mulps_r2r(XMM0, XMM1); // scale elements with a
        movups_m2r(*(((sse_t *) y)+i), XMM2); // next 4 y's
        addps_r2r(XMM1, XMM2); // add a*x to y
        movups_r2m(XMM2, *(((sse_t *) y)+i)); // store 4 y's
    }
}
In the first example (Apple version of Power PC)

- $Si = \text{vec\_ld}(0,\text{ptr}0)$ loads 4 words beginning at location $\text{ptr}0$ (offset = 0) into $Si$.
- $\text{vec\_st}(Si,0,\text{ptr}1)$ stores 4 words starting at location $\text{ptr}1$ (offset = 0) from $Si$.
- $\text{vec\_madd}(Si,Sj,Sk)$ multiplies $Si$ times $Sj$ and adds this result to $Sk$, result is again stored in $Sk$.

In the second example (Intel SSE)

- $\text{movups\_m2r}(ptr,\text{XM}1)$ loads 4 words from $ptr$ to $\text{XM}1$
- $\text{mulps\_r2r}(\text{XMM}0, \text{XMM}1)$ scales $\text{XMM}1$ by $\text{XMM}0$
- $\text{add\_r2r}(\text{XMM}1, \text{XMM}2)$ adds $a*x + y$
- $\text{movups\_r2m}(\text{XMM}2, \text{ptr})$ stores 4 $\text{XMM}2$s into $ptr$
Because of technical difficulties of this sort, the BLAS (Basic Linear Algebra Subprograms) were proposed


These BLAS assume Fortran linkages and column ordering. Column ordering means $A(NC,NR)$ has $A(i+1,j)$ stored immediately after $A(i,j)$. All parameters are passed by address.
**BLAS level 1, vector-vector operations**

<table>
<thead>
<tr>
<th>Reduction operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \leftarrow x \cdot y$</td>
</tr>
<tr>
<td>$s \leftarrow \max{</td>
</tr>
<tr>
<td>$s \leftarrow |x|_2$</td>
</tr>
<tr>
<td>$s \leftarrow \sum_i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector to vector transformations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y \leftarrow x$</td>
</tr>
<tr>
<td>$x \leftrightarrow y$</td>
</tr>
<tr>
<td>$y \leftarrow \alpha \cdot x$</td>
</tr>
<tr>
<td>$y \leftarrow \alpha \cdot x + y$</td>
</tr>
</tbody>
</table>
BLAS-1 (continued):

Generate and apply Givens rotations:

<table>
<thead>
<tr>
<th>Compute rotation:</th>
</tr>
</thead>
</table>
| \[
\begin{pmatrix}
  c & s \\
  -s & c
\end{pmatrix}
\begin{pmatrix}
  a \\
  b
\end{pmatrix}
\rightarrow
\begin{pmatrix}
  r \\
  0
\end{pmatrix}
\]
| \( c, s \ni r = \sqrt{a^2 + b^2} \) _ROTG

<table>
<thead>
<tr>
<th>Apply rotation:</th>
</tr>
</thead>
</table>
| \[
\begin{pmatrix}
  x \\
  y
\end{pmatrix}
\leftarrow
\begin{pmatrix}
  c & s \\
  -s & c
\end{pmatrix}
\begin{pmatrix}
  x \\
  y
\end{pmatrix}
\]   _ROT
### Type prefixes/suffixes:

<table>
<thead>
<tr>
<th>Prefixes:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
</tr>
<tr>
<td><strong>D</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td><strong>Z</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suffixes:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
</tbody>
</table>
BLAS level 2, matrix-vector operations

<table>
<thead>
<tr>
<th>Matrix times Vector</th>
<th>general</th>
<th>general band</th>
<th>general hermitian</th>
<th>hermitian banded</th>
<th>hermitian packed</th>
<th>general symmetric</th>
<th>symmetric banded</th>
<th>symmetric packed</th>
<th>triangular</th>
<th>triangular banded</th>
<th>triangular packed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{x} \leftarrow \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$</td>
<td>_GEMV</td>
<td>_GBMV</td>
<td>_HEMV</td>
<td>_HBMV</td>
<td>_HPMV</td>
<td>_SYMV</td>
<td>_SBMV</td>
<td>_SPMV</td>
<td>_TRMV</td>
<td>_TBMV</td>
<td>_TPMV</td>
</tr>
<tr>
<td>$\mathbf{x} \leftarrow \mathbf{A} \mathbf{x}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BLAS-2, matrix-vector operations (cont.)

<table>
<thead>
<tr>
<th>Triangular solve:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow A^{-1}x$</td>
</tr>
<tr>
<td>$A$ triangular $_\text{TRSV}$</td>
</tr>
<tr>
<td>$A$ triangular banded $_\text{TBSV}$</td>
</tr>
<tr>
<td>$A$ triangular packed $_\text{TPSV}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank one and rank two updates:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \leftarrow \alpha xy^T + A$</td>
</tr>
<tr>
<td>$A$ general $_\text{GER}$</td>
</tr>
<tr>
<td>$A \leftarrow \alpha xx^* + A$</td>
</tr>
<tr>
<td>$A$ general hermitian $_\text{HER}$</td>
</tr>
<tr>
<td>$A$ hermitian packed $_\text{HPR}$</td>
</tr>
<tr>
<td>$A \leftarrow \alpha(xy^* + yx^*) + A$</td>
</tr>
<tr>
<td>$A$ gen. Hermitian $_\text{HER2}$</td>
</tr>
<tr>
<td>$A$ hermitian packed $_\text{HPR2}$</td>
</tr>
</tbody>
</table>
BLAS-2, matrix-vector operations (cont.)

\[ A \leftarrow \alpha x x^T + A \]

general symmetric \_SYR

symmetric packed \_SPR

\[ A \leftarrow \alpha (x y^T + y x^T) + A \]

gen. symmetric \_SYR2

symmetric packed \_SPR2
BLAS level 3, matrix-matrix operations

<table>
<thead>
<tr>
<th>Matrix product:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong> ←</td>
</tr>
<tr>
<td>[ \alpha A \cdot B + \beta C ]</td>
</tr>
<tr>
<td>general</td>
</tr>
<tr>
<td>symmetric</td>
</tr>
<tr>
<td>symmetric</td>
</tr>
<tr>
<td>hermitian</td>
</tr>
<tr>
<td>hermitian</td>
</tr>
<tr>
<td><strong>B</strong> ←</td>
</tr>
<tr>
<td>[ \alpha A \cdot B ]</td>
</tr>
<tr>
<td>triangular</td>
</tr>
<tr>
<td>triangular</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank k update:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C</strong> ←</td>
</tr>
<tr>
<td>[ \alpha A \cdot A^T + \beta C ]</td>
</tr>
<tr>
<td>SYRK</td>
</tr>
<tr>
<td><strong>C</strong> ←</td>
</tr>
<tr>
<td>[ \alpha A \cdot A^H + \beta C ]</td>
</tr>
<tr>
<td>HERK</td>
</tr>
<tr>
<td><strong>C</strong> ←</td>
</tr>
<tr>
<td>[ \alpha (A \cdot B^T + B \cdot A^T) + \beta C ]</td>
</tr>
<tr>
<td>SYRK2</td>
</tr>
<tr>
<td><strong>C</strong> ←</td>
</tr>
<tr>
<td>[ \alpha (A \cdot B^H + B \cdot A^H) + \beta C ]</td>
</tr>
<tr>
<td>HERK2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Triangular solve for multiple r.h.s.:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>B</strong> ←</td>
</tr>
<tr>
<td>[ \alpha A^{-1} \cdot B ]</td>
</tr>
<tr>
<td>triangular</td>
</tr>
<tr>
<td>_TRSM</td>
</tr>
</tbody>
</table>
BLAS are the low level procedures (optimized) used by

- LAPACK, ScaLAPACK - Linear Algebra (e.v’s, linear eqs.)
- LINPACK (linear equation solvers, Argonne, BLAS-1)
- NAG library (Numerical Algorithms Group, UK)
- IMSL library (Houston)
- NL2SOL (D. M. Gay, BTL)
- FNLIB (W. Fullerton, BTL)
- QUADPACK (Leuven)

see: netlib.org

There are C versions of BLAS, too. The most optimal versions are for FORTRAN libraries. Hence, we do some C ↔ FORTRAN communication.
A quick summary of communication between FORTRAN and C.

- FORTRAN passes all arguments to procedures by **address**

  program address
c
c  this is OK:
c
c  real x
call subr(x)
print *," x=" , x
stop
def
d subroutine subr(y)
real y
y = 3.14159
return
def
C passes all arguments to procedures by **value**, except arrays, which are passed by **address**.

```c
#include <stdio.h>
main()
{
    float x,y,*z;
    void subrNOK(),subrOK();
    /* this is not OK: */
    subrNOK(x);
    printf(" x= %e\n",x);
    /* this is OK: */
    subrOK(&y,z);
    printf(" y = %e, z[0] = %e\n",y,*z);
}
```
this one is **incorrect**

```c
void subrNOK(float a)
{  a = 3.14159; }
```

this one is **correct**

```c
void subrOK(float *b, float *c)
{
    *b = 3.14159;
    c[0] = 2.71828;
}
```
FORTRAN allows dynamically dimensioned arrays.

```fortran
program dims
  c array x is linear when declared
  real x(9)
  call subr(x)
  print *, x
  stop
end

subroutine subr(x)
  real x(3,3)
  do i=1,3
    do j=1,3
      x(j,i) = float(i+j)
    enddo
  enddo
  return
end
```

- Fortran index counts from 1, C indexes from 0.
- C does not allow dynamically dimensioned arrays.
- C uses "&x" to indicate the **address** of x.
- C uses "*px" to indicate a **value** at addr. px → x. thus
  
  ```
  px = &x;
y = *px;
  /* is the same as */
y = x;
  ```

- Character strings: e.g. calling DGEMM

  ```
  char ta = 'N', tb = 'N';
  alpha = 1.0; beta = 0.0;
  dgemm_(&ta, &tb, &nra, &ncb, &nca, &alpha,
          a, &n, b, &n, &beta, c, &n);
  ```
To call a Fortran procedure from C:

```c
float x,*y,z, fcn_(); /* NOTE underscore */
void subr_(); /* also underscore */
subr_(&x,y);
z = fcn_(&x);
...
```

```fortran
subroutine subr(r,s)
  real r,s(*)
  r    = 3.14159
  s(1) = 2.71828
  return
end

real fcn(r)
real r
fcn = exp(r**1.5)
return
end
```
To call a C procedure from Fortran:

```fortran
real x,y
call subr(x,y)
...
```

```c
void subr_(r,s) /* note underscore */
float *r,*s;
{
  *r = 3.14159;
  *s = 2.71828;
}
```

gcc ↔ gfortran no longer need supporting libraries:
for example, C calls Fortran: `libg2c.a` or `libg2c.so`

However, you will need the appropriate libraries
```
gfortran -O3 -c myFTNcode.f
gcc -O3 -fopenmp myCcode.c myFTNcode.o -lblas or,
gcc -O3 myCcode.c myFTNcode.o -lblas -lgomp
```
Our example: SGEFA is the LINPACK benchmark for the Top 500.

Figure: Gaussian elimination diagram.
Here, again, is the SGEFA code:

```c
#define am(p,q) (a+p+q*la)
void sgefa(a,la,n,ip,info)
float *a;
int n,la,*ip,*info;
{
    /* parallel C version of SGEFA 18/4/2000 */
    int k,kp1,l,nm1,nmk,nmk1,one,isamax_();
    float t;
    void msaxpy();
    void sswap_();
    void sscal_();

    *info = 0;
    nm1 = n-1; one = 1;
```
for(k=0;k<nm1;k++){
    kp1 = k+1; nmk = n-k; nmk1 = nmk - 1;
    l = isamax_(&nmk,am(k,k),&one) + k - 1;
    ip[k] = l;
    if((*am(l,k))==0.0){
        *info=k; return;
    }
    if(l!=k){
        t = *am(l,k); *am(l,k) = *am(k,k); *am(k,k) = t;
    }
    t = -1.0/(*am(k,k));
    sscal_(&nmk1,&t,am(kp1,k),&one);
    if(l!=k){
        sswap_(&nmk1,am(l,kp1),&la,am(k,kp1),&la);
    }
    msaxpy(nmk1,nmk1,am(k,kp1),n,am(kp1,k),am(kp1,kp1));
}
void
msaxpy(int nr, int nc, float *a, int n, float *x, float *y)
{ /* wpp 29/01/2003 */
    int i, j;
    #pragma omp parallel shared(nr, nc, a, x, y) private(i, j)
    #pragma omp for schedule(static, 32) nowait
    for(j=0; j<nc; j++){
        for(i=0; i<nr; i++){
            *ym(i, j) += a[j*n]*x[i];
        }
    }
}
It is very tough to beat libraries. For example, SGETRF from LAPACK: Let’s compare my last example SGEFA using OMP:

Figure: Comparing libraries vs. me

bulldozer: export OMP_NUM_THREADS=$N_T$, and gomp
Here is the double version: DGETRF from LAPACK compared to DGEFA.

Figure: Comparing libraries vs. me
References:

- GCC pragma directives: https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html