NTNU – Trondheim
Norwegian University of
Science and Technology

Cache optimizations: loop tiling & vector operations

Jan.Christian.Meyer@ntnu.no

While we are close to the metal

- We started with message passing
  - Data travels from process to process
    (optionally, from computer to computer)
  - O/S and interconnect are invisible, but affect performance
- We've gone through threading in two flavors
  - Data travels from local memory to local memory
  - Caching and instruction selection are invisible, but affect performance
- Before we move on, I'd like to
  - demonstrate some kind-of explicit cache manipulation
  - demonstrate a little bit of manual instruction selection

We need something to compute

- The calculations we've been working with so far don't make the very best demonstration
  - They need too many memory accesses per operation
  - More on that later
- General Matrix-Matrix multiplication works well
  - *"gemm"* among its friends
- There are some very fancy algorithms to do this
  - We'll just use the naive one

NTNU – Trondheim
Norwegian University of Science and Technology

Vector (dot) product

(in case you forgot)

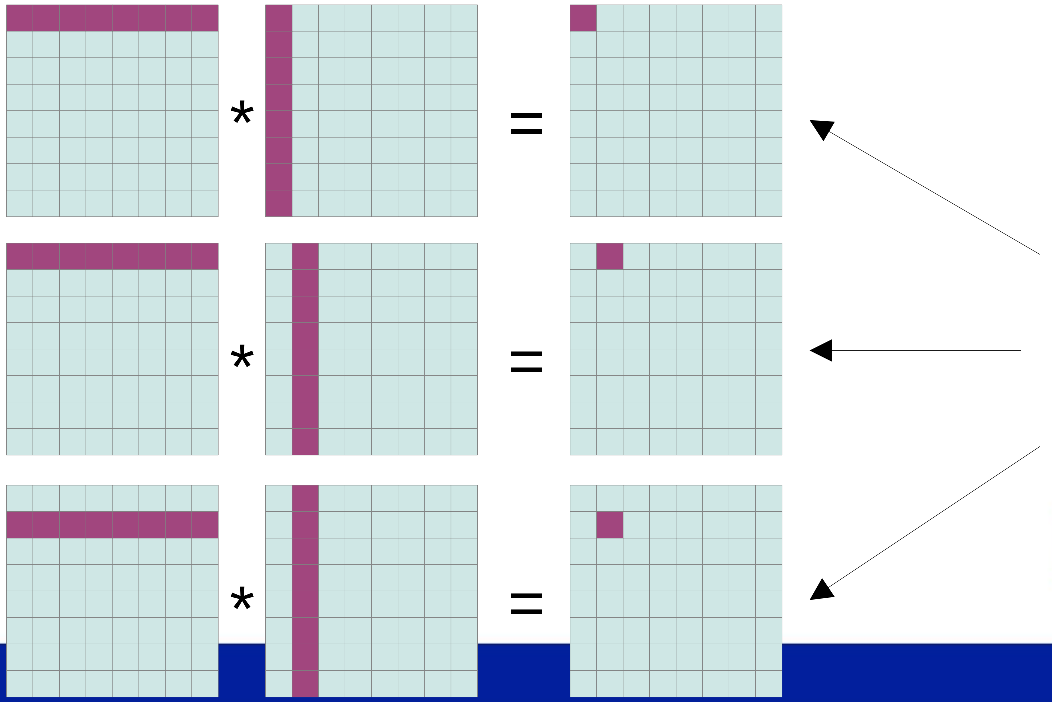
- The product of two vectors is the sum of the products of their elements
  - Row/column notation isn't really important for a pair of vectors, I'll just draw it like this anyway

$$\begin{bmatrix} a0 & a1 & a2 & a3 \end{bmatrix} * \begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \end{bmatrix} = a0b0 + a1b1 + a2b2 + a3b3$$

(One pair of vectors produces 1 number)

Matrix (dot) product

- When we've got two matrices A and B, get the third one (C) by multiplying all pairs of vectors
  - The width of A has to equal the height of B, but we can simplify my demonstration by just thinking of square matrices today



The naive procedure

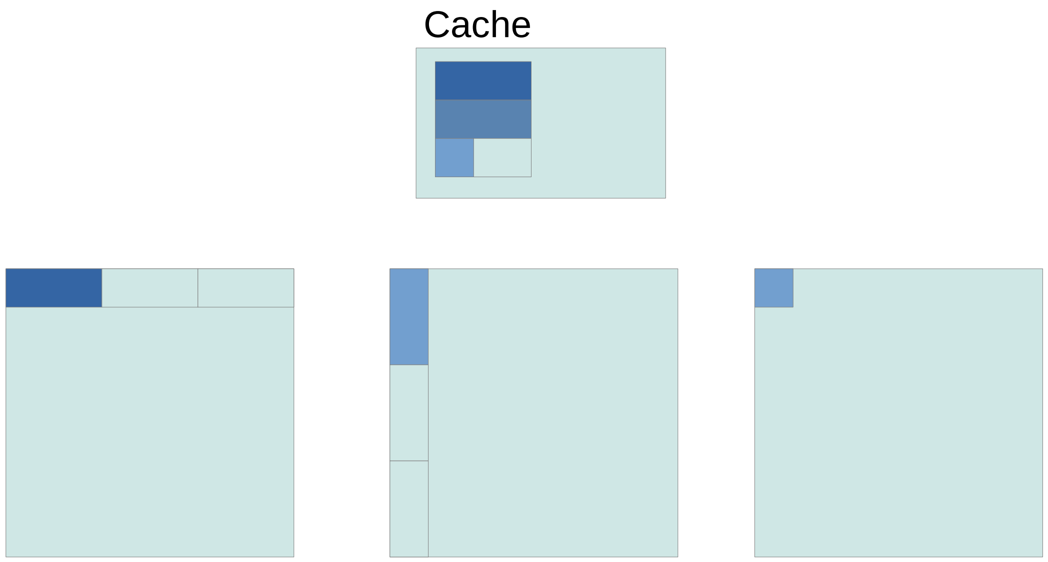
- For NxN matrices A,B,C, this translates into triple-nested loops:

```
for ( int i=0; i<N; i++ )
    for ( int j=0; j<N; j++ )
        for ( int k=0; k<N; k++ )
            C(i,j) += A(i,k) * B(k,j);
```

- The number of elements are $O(N^2)$
- The number of operations are $O(N^3)$
- The same data appear in several different products

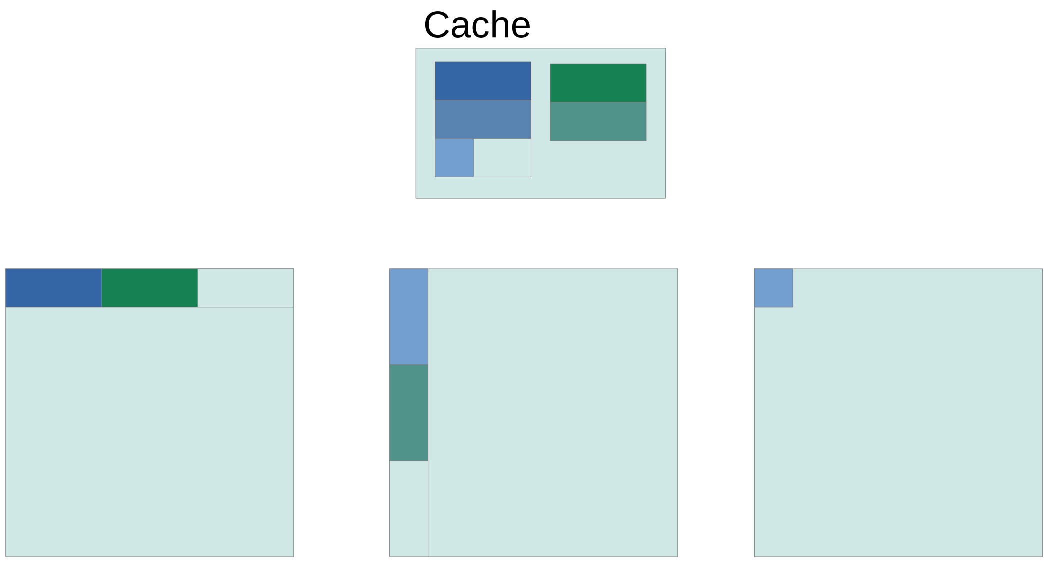
What happens in cache?

- When the matrices grow big enough, the first cache lines we load must be evicted before they can be re-used



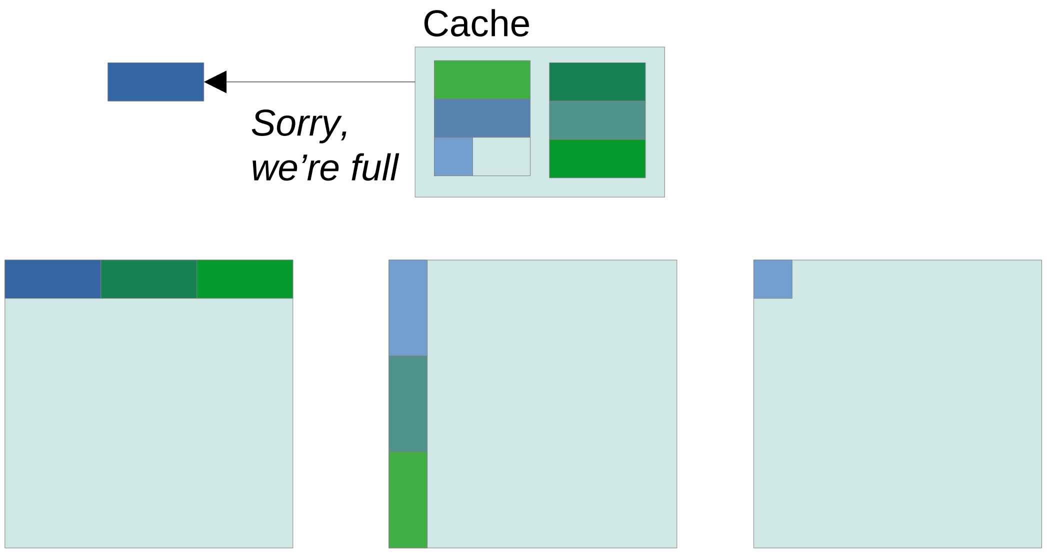
What happens in cache?

- When the matrices grow big enough, the first cache lines we load must be evicted before they can be re-used



What happens in cache?

- By the end, the first thing we loaded must be replaced
  - That’s a shame, because we’ll need it again in a moment

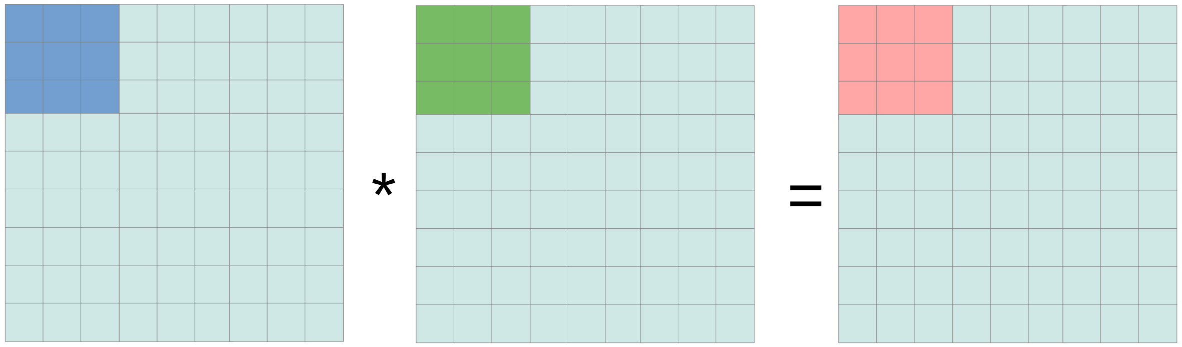


Improving re-use

- For matrices that are small enough to fit entirely in cache, this is not a problem
  - Obviously, because cache space never runs out then
- What can we do?
  - The same, naive algorithm works fine for smaller matrices
  - If we can express a large matrix product as a sum of smaller ones, we can use it to greater effect

Tiling

- Suppose we just multiply some smaller sub-matrices to begin with:

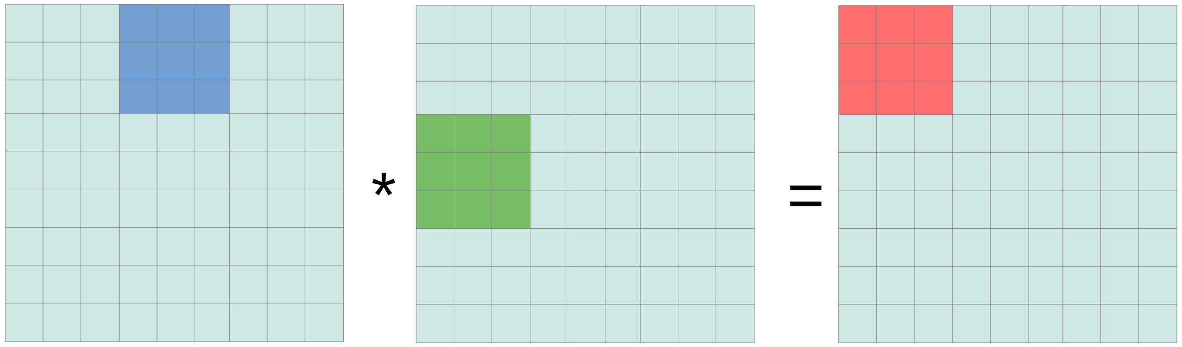


- Each element in the end product will only contain three out of nine products that go into the total
  - That's ok, we can add them later
  - Additions don't care what sequence you add in

NTNU – Trondheim
Norwegian University of Science and Technology

Tiling, stage 2

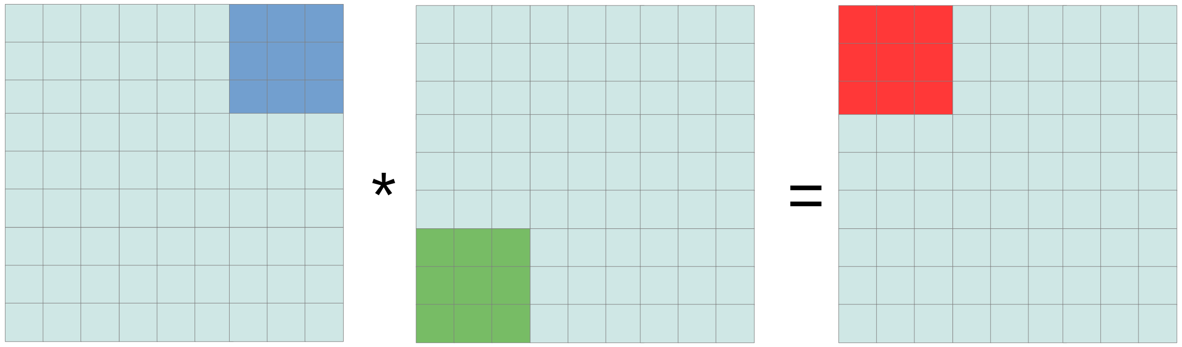
- We can move the sub-matrices we're multiplying in the same pattern as the single elements



- Now we have six in nine parts out of the results in the C-matrix tile

Tiling, in the end

- When our moving tiles reach the end of the matrices, the result-matrix tile is complete



- For every product of 3x3 tiles, we've used every value in the tiles as part of 3 different partial products
- This should be great for cache, as long as the tiles fit

Let’s try it!

- Today’s code archive contains three matrix multipliers
  - dgemm_naive.c (produces a file called ‘correct.dat’)
  - dgemm_tiled.c (produces a file called ‘tiled.dat’)
  - dgemm_vectorized.c (produces ‘vectorized.dat’)

  - The programs time their own execution
  - The ‘correct’ file is for comparing the other two results
  - They won’t necessarily be binary-identical, but there’s a program ‘compare.c’ which tests them for equality to within 10^{-12}

Vector registers

- We've talked about how vector registers can hold more than 1 value at a time
  ...and apply the same operation to all of them simultaneously
  - SIMD execution, if you recall
- Vector registers get a little architecture-specific
  - They're really best left for a compiler to generate
- Compiler vector detection can fail for many reasons
  - When it doesn't vectorize automatically for you, it can be useful to do by hand
  - I'll demonstrate how to program the Intel vector registers explicitly

Intrinsics

- We *could* do this with direct assembly programming
  - It's error prone and quirky, though
- Because vector instructions can be useful in higher level languages as well, they're supported by *intrinsics*
  - Constructs that behave like function calls, but the compiler can recognize as shorthand notation for assembly instructions
- We'll only need intrinsics for SSE2 instructions
  - Introduced with the Pentium4 instruction set
  - There are newer additions, but these have vectors that hold pairs of double precision floating point values
  - That's enough for demonstration
- Enable with

  #include <emmintrin.h>

Data types & memory allocation

- __mm128d my_variable;
  - This is like a declaration, but the type stands for a 128-bit SIMD register, so it can hold two 64-bit doubles
  - It’s not actually a 1-1 mapping with a SIMD register, you can declare more of these variables than there are registers
  - The point is that it’s a blob of bytes which can be put into such a register in an instant, and fit there
- _mm_malloc (size, alignment);
  - This is like ‘malloc’, but the ‘alignment’ argument gives a number that evenly divides the starting address of the allocation
  - Vector registers load faster when the addresses they load are clean multiples of the register size
  - Since we have 16-byte (128bit) values, we’ll use 16

Load/store operations

- In order to transfer data from memory into a SIMD register, you can write

  __m128d my_two_doubles = _mm_load_pd (&two_doubles);

  (provided that the address of 'two_doubles' is 16-byte aligned)

  You can also load two copies of one double:

  __m128d two_copies_of_x = _mm_load_pd1 (&x);
- You can also move data from SIMD registers to memory

  _mm_store_pd (my_two_doubles, &two_doubles);

  (still assuming aligned addresses)
- These are RISC-style instructions
  - Explicit loading and storing, other operations only combine registers

Multiplication and addition

- Pairwise addition of two SIMD registers works thus:

  __m128d aplusc_and_bplusd = _mm_add_pd (ab, cd);

- Pairwise multiplication is similar

  __m128d ac_and_bd = _mm_mul_pd (ab, cd);

- These are the operations we need for dgemm
  - There are many more…
  - Un-aligned loading and storing is possible (but slower)
  - SSE2 can also do 4-long vectors of 32bit floats
  - SSE3, AVX, AVX2, AVX512 have extended the op. set
  - Consult the intrinsics guide for a full reference: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

NTNU – Trondheim
Norwegian University of
Science and Technology

Vectorizing 2x2 tiles

- If we write out the 2x2 matrix product

  C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0)
  C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1)
  C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0)
  C(1,1) = A(1,0)*B(1,0) + A(1,1)*B(1,1)

  we can notice that some terms appear in pairs:

  C(0,0) = **A(0,0)***B(0,0) + **A(0,1)***B(1,0)
  C(0,1) = **A(0,0)***B(0,1) + **A(0,1)***B(1,1)
  C(1,0) = **A(1,0)***B(0,0) + **A(1,1)***B(1,0)
  C(1,1) = **A(1,0)***B(0,1) + **A(1,1)***B(1,1)

NTNU – Trondheim
Norwegian University of
Science and Technology

Vectorizing 2x2 tiles

- The other factors with those pairs are consecutive

  C(0,0) = A(0,0)***B(0,0)** + A(0,1)***B(1,0)**
  C(0,1) = A(0,0)***B(0,1)** + A(0,1)***B(1,1)**
  C(1,0) = A(1,0)***B(0,0)** + A(1,1)***B(1,0)**
  C(1,1) = A(1,0)***B(0,1)** + A(1,1)***B(1,1)**

  in other words, we can handle the products with six 2-vectors:

  [A(0,0) A(0,0)] x [B(0,0) B(0,1)]
  [A(0,1) A(0,1)] x [B(0,1) B(1,1)]
  [A(1,0) A(1,0)] x [B(0,0) B(1,0)]
  [A(1,1) A(1,1)] x [B(0,1) B(1,1)]

NTNU – Trondheim
Norwegian University of
Science and Technology

Vectorized dgemm

- The third version of our matrix multiplier uses this notation in order to further speed up our tiled routine a little bit

  ...but only for 2x2 tiles, because we only use 2-length vectors

- Note that while your compiler will usually vectorize a plain loop like

```
for ( int i=0; i<N; i++ )
    c[i] += a[i]*b[i];
```

  it doesn't as easily recognize the vector-potential we uncovered in the tiling routine

  – That's why I'm showing you it can be done by hand

NTNU – Trondheim
Norwegian University of Science and Technology

Reality check

- The tiling effect is much more prominent than the vector one
- We only did 1 level of tiling
  - As you can probably see, it's possible to do tiled multiplication within the tiles as well
  - At optimum, we have a tile granularity for each cache level, so that L3-size tiles are multiplied using L2-size tiles, which are multiplied by L1-size tiles that can be tiled with vectors
  - Too much typing for a 45 minute lecture, though
- Honestly, the easiest way to do it is to use someone else's highly tuned library function
  - Look into ATLAS, OpenBLAS or MKL if you want to multiply a lot of matrices
  - Now we know roughly how they work, though