Optimization techniques

Optimization training at CINES

Adrien Cassagne

adrien.cassagne@inria.fr

Inria
INVENTEURS DU MONDE NUMÉRIQUE

CINES
Centre Informatique National
de l'Enseignement Supérieur

2015/12/01

Contents

Contents

1 Scalar optimizations
- Pre-processing
- Avoiding branch instructions
- Avoiding divisions
- Special functions
- Memory accesses
- Cache blocking
- Inlining
- Compiler options

2 In-core parallelism

3 Multi-core optimizations

Pre-compute things when it is possible

- We can divide a computational code in 3 parts
  1. Pre-processing: allocations, initializations, reading of inputs
  2. Solver or massively computational part
  3. Post-processing: writing of outputs, deallocations
- In many cases the solver part is the most time consuming
- When it is possible, we have to pre-compute things in the pre-processing part
  - The idea is to reduce the total number of operations in the code

Pre-processing example

```cpp
void main()
{
  // pre-processing part
  int n = 1000;
  float *A, *B, *C, *D;
  A = new float[n]; // in
  B = new float[n]; // in
  C = new float[n]; // in
  D = new float[n]; // out
  randomInit(A, n); randomInit(B, n); randomInit(C, n);
  for(int i = 0; i < n; i++)
    D[i] = 0;

  // solver or computational part
  for(int j = 0; j < n; j++)
    for(int i = 0; i < n; i++)
      D[i] = D[i] + (A[i] + B[i]) * C[j];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
  delete[] D;
}
```

Simple computational code, not optimized

- The total number of operations (flops) is $n \times n \times 3$

Pre-processing example

```
void main()
{
  // pre-processing part
  int n = 1000;
  float *A, *B, *C, *D, *AB;
  A  = new float[n]; // in
  B  = new float[n]; // in
  AB = new float[n]; // pre-processing
  C  = new float[n]; // in
  D  = new float[n]; // out
  randomInit(A, n); randomInit(B, n); randomInit(C, n);
  for(int i = 0; i < n; i++) {
    D [i] = 0;
    AB[i] = A[i] + B[i]; // we pre-compute A + B
  }

  // solver or computational part
  for(int j = 0; j < n; j++)
    for(int i = 0; i < n; i++)
      D[i] = D[i] + AB[i] * C[j];

  // post-processing part ...
}
```

Simple computational, code optimized

- The total number of operations (flops) is $n + (n \times n \times 2)$
- Can we do better than that?

Pre-processing example

```
void main()
{
  // pre-processing part
  int n = 1000;
  float *A, *B, *C, *D, *AB;
  A  = new float[n]; // in
  B  = new float[n]; // in
  C  = new float[n]; // in
  D  = new float[n]; // out
  randomInit(A, n); randomInit(B, n); randomInit(C, n);
  float sumC = 0;
  for(int i = 0; i < n; i++)
    sumC += C[i]; // we pre-compute the sum of C[O] + C[1] + ... + C[n -1]

  // solver or computational part
  for(int i = 0; i < n; i++)
    D[i] = (A[i] + B[i]) * sumC;

  // post-processing part ...
}
```

Simple computational code, well optimized

- The total number of operations (flops) is $3n$

Branch instructions

- Branch instructions (alias `if`, `switch`, etc) create bubbles in the processor pipeline
- The pipeline can't be fully filled
- We have to try to reduce the use of this kind of instructions

```c
void main()
{
  // pre-processing part ...

  // solver or computational part
  // there is an implicit branch instruction in the loop
  for(int i = 0; i < n; i++) {
    // compiler generate a branch instruction here
    if(i >= 1 && i < n -1) {
      // compiler generate an other branch instruction here
      switch(i % 4) {
        case 0: B[i] = A[i] * 0.3333f;
        case 1: B[i] = A[i] + 1.3333f;
        case 2: B[i] = A[i] - 0.7555f;
        case 3: B[i] = A[i] * 1.1111f;
        default: break;
      }
    }
  }

  // post-processing part..
}
```

Computational code with branch instructions

Reducing the number of branch instructions

```
void main()
{
  // pre-processing part
  const int n = 40000002;
  float *A, *B;
  A = new float[n]; // in
  B = new float[n]; // out
  randomInit(A, n);
  struct timeval t1, t2;

  // solver or computational part
  for(int i = 1; i < n -1; i += 4) {
    B[i+0] = A[i+0] + 1.3333f;
    B[i+1] = A[i+1] - 0.7555f;
    B[i+2] = A[i+2] * 1.1111f;
    B[i+3] = A[i+3] * 0.3333f;
  }

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Computational code with less branch instructions

- In the previous code with have replaced the `switch` by a 4-unrolling loop
- And we have replaced the `if` by starting the loop with `1` and finishing it with `n -1`

Division

- Here is the cost of the main operations:
  - `add`: 1 CPU cycle
  - `sub`: 1 CPU cycle
  - `mul`: 1 CPU cycle
  - `div`: $\approx$ 20 CPU cycles
- As we can see, a division is very expensive compared to a multiplication
- It is much better to compute the inverse number and multiply by it!
- Be careful, when we multiply by inverse we lose some precision in the calculation

Division example

```
void main()
{
  // pre-processing part
  int n = 1000;
  float *A, *B;
  A = new float[n]; // in
  B = new float[n]; // out
  randomInit(A, n);

  // solver or computational part
  for(int i = 0; i < n; i++)
    B[i] = A[i] / 3.f;

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Simple computational code with divisions

- Theoretical number of cycles: $n \times 20$

Division example

```
void main()
{
  // pre-processing part
  int n = 1000;
  float *A, *B;
  A = new float[n]; // in
  B = new float[n]; // out
  randomInit(A, n);
  float inv3 = 1.f / 3.f;

  // solver or computational part
  for(int i = 0; i < n; i++)
    B[i] = A[i] * inv3;

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Simple computational code without divisions

- Theoretical number of cycles: $20 + (n \times 1)$

Special functions

- Here is the cost of the main special functions:
  - `pow`: very expensive, the number of cycles depends on the input
  - `sqrt`: ≈ 30 CPU cycles
  - `rsqrt`: 1 CPU cycle
  - `cos`: very expensive, the number of cycles depends on the input
  - `sin`: very expensive, the number of cycles depends on the input
  - `tan`: very expensive, the number of cycles depends on the input
- `rsqrt` take 1 cycle!
  - This is very surprising
  - In fact there are hardware pre-compute tables in today CPUs
  - The CPU simply returns the nearest value we need
- `pow`, `sqrt`, `cos`, `sin` and `tan` are very expensive try to not use them in the solver part of the code
- If it is not possible, at least try to reduce the number of calls

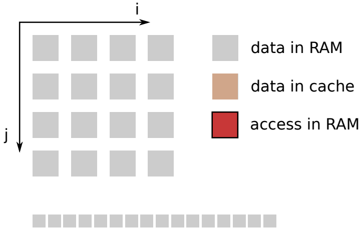
Memory accesses

- When the code is memory bound, we have to carefully consider data structures and data accesses
- Memory bandwidth is slow compared to CPU computational capacity
  - There are some mechanisms to reduce this memory lack: pre-fetching data
  - Remember, cache accesses are done per line of words (and not per words)
  - So, it is very interesting to work on stream data
  - Also, we have to reduce direct accesses in RAM and maximize accesses in cache

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices

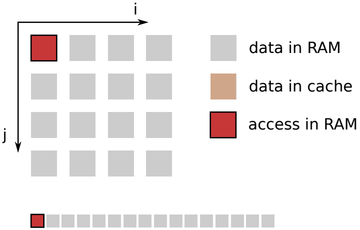


Logical and hardware view of matrix in memory

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices



data in RAM

data in cache

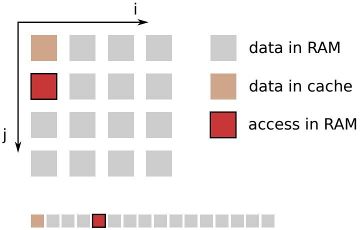
access in RAM

Logical and hardware view of matrix in memory

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices



data in RAM

data in cache

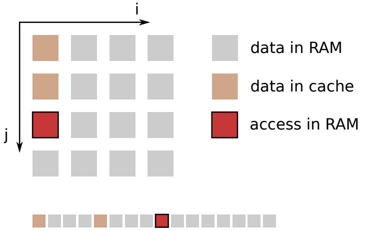
access in RAM

Logical and hardware view of matrix in memory

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices

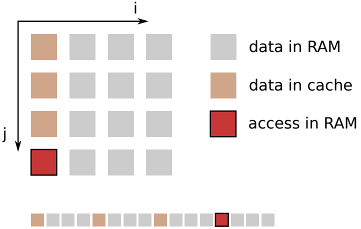


Logical and hardware view of matrix in memory

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices



data in RAM

data in cache

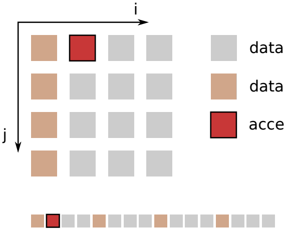
access in RAM

Logical and hardware view of matrix in memory

Memory accesses example

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices



⬜ data in RAM

🟫 data in cache

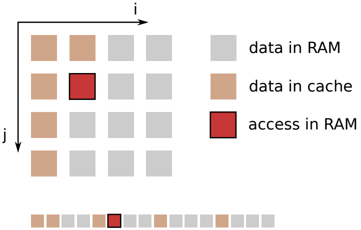
🟥 access in RAM

Logical and hardware view of matrix in memory

Memory accesses example

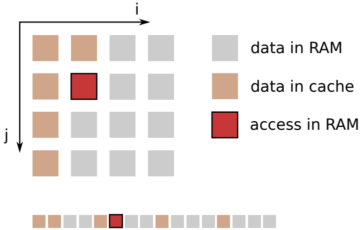
```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int i = 0; i < n; i++) // column
    for(int j = 0; j < n; j++) // row
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices



Logical and hardware view of matrix in memory

Memory accesses example



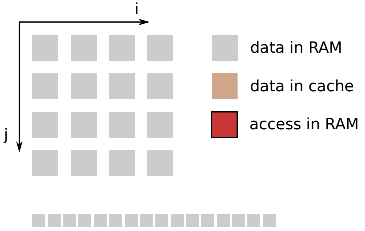
Logical and hardware view of matrix in memory

- In this implementation data accesses are not contiguous in memory
- There is a 4-stride between each access

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)



⬜ data in RAM

🟫 data in cache

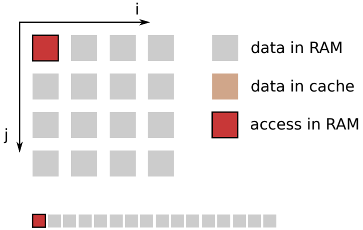
🟥 access in RAM

Logical and hardware view of matrix in memory

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)

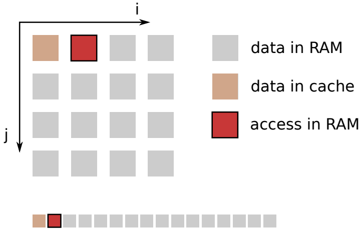


Logical and hardware view of matrix in memory

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)

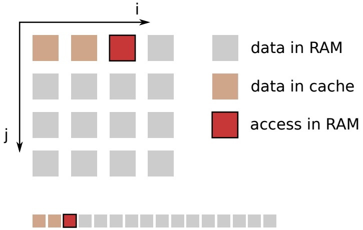


Logical and hardware view of matrix in memory

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)

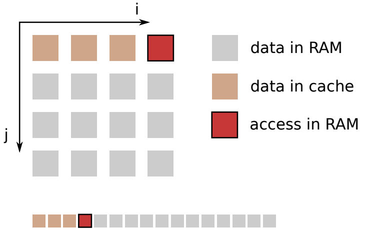


Logical and hardware view of matrix in memory

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)

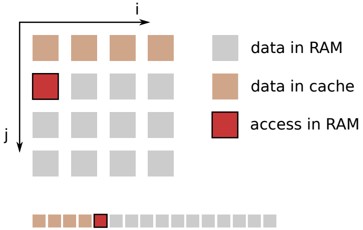


Logical and hardware view of matrix in memory

Memory accesses example: solution

```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)

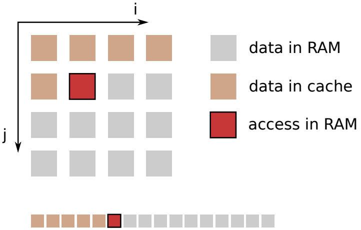


Logical and hardware view of matrix in memory

Memory accesses example: solution

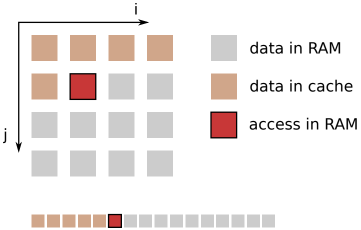
```
void main()
{
  // pre-processing part
  int n = 4;
  float *A, *B, *C;
  A  = new float[n*n]; // in
  B  = new float[n*n]; // in
  C  = new float[n*n]; // out
  randomInit(A, n); randomInit(B, n);

  // solver or computational part
  for(int j = 0; j < n; j++) // row
    for(int i = 0; i < n; i++) // column
      C[i + j*n] = A[i + j*n] + B[i + j*n];

  // post-processing part
  delete[] A;
  delete[] B;
  delete[] C;
}
```

Adding square matrices (solution)



Logical and hardware view of matrix in memory

Memory accesses example: solution



Logical and hardware matrix memory view

- In this implementation data accesses are perfectly contiguous in memory
  - Loaded cache lines are well used
  - Performance will be better
- We have simply switched the i-loop with the j-loop
- In this case (addition of matrices) there is no data reuse
  - This code is very very memory bound!

Cache blocking technique

- In many cases, some data can be reused!
- Let's take an example with a code working on a 2D grid

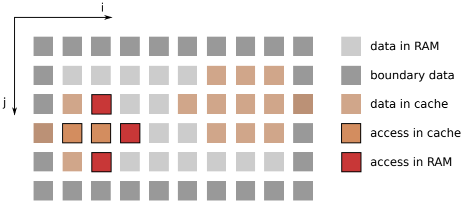
```
void main()
{
  // pre-processing part
  int cols = 10, rows = 6;
  float *A = new float[cols*rows]; // in
  float *B = new float[cols*rows]; // out
  randomInit(A, cols*rows);

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] +
                      A[(i   ) + (j   )*cols] +
                      A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Stencil code

Cache blocking technique

```
void main()
{
  // pre-processing part ...

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] + // left, right
                      A[(i   ) + (j   )*cols] +                           // center
                      A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];  // top, bottom

  // post-processing part ...
}
```

Stencil code

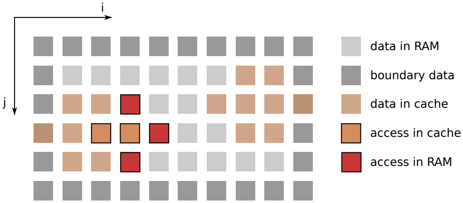


Logical 2D grid memory view

Cache blocking technique

```
void main()
{
  // pre-processing part ...

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] + // left, right
                      A[(i   ) + (j   )*cols] +                           // center
                      A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];  // top, bottom

  // post-processing part ...
}
```

Stencil code

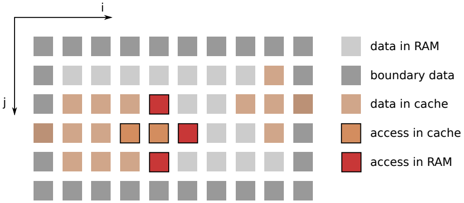


Logical 2D grid memory view

Cache blocking technique

```
void main()
{
  // pre-processing part ...

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] + // left, right
                      A[(i   ) + (j   )*cols] +                          // center
                      A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];  // top, bottom

  // post-processing part ...
}
```

Stencil code

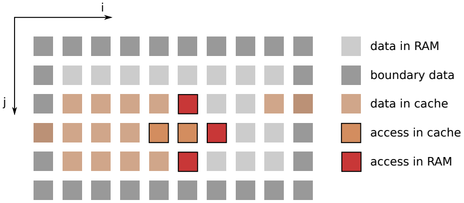


Logical 2D grid memory view

Cache blocking technique

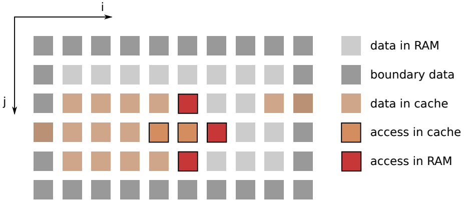
```
void main()
{
  // pre-processing part ...

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] + // left, right
                      A[(i   ) + (j   )*cols] +                           // center
                      A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];  // top, bottom

  // post-processing part ...
}
```

Stencil code



Logical 2D grid memory view

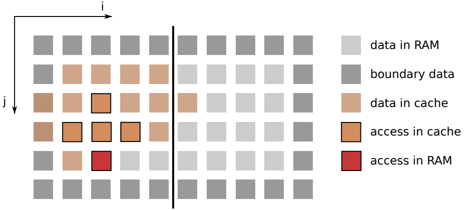
Cache blocking technique



Logical 2D grid memory view

- Each time we have 3 accesses in the RAM and 2 accesses in the cache
- Can we do better? Can we decrease the number of RAM accesses?
  - Yes, we can with the cache blocking technique!
  - The idea is to modify the data accessing manner in order to maximize data re-utilization

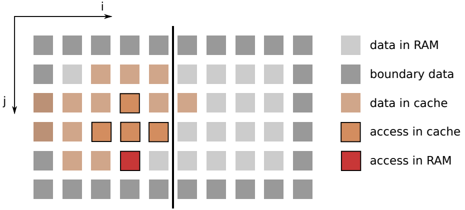
Cache blocking technique: example



Logical 2D grid memory view

- With the cache blocking technique we reduce the number of RAM accesses
  - It remains just 1 access in RAM (sometimes 2)!
  - We cut the grid in different blocks (with a vertical separation here)

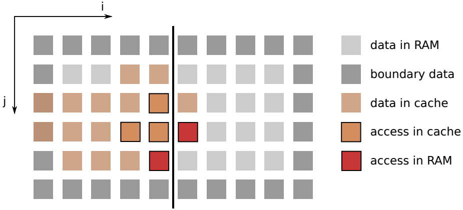
Cache blocking technique: example



Logical 2D grid memory view

- With the cache blocking technique we reduce the number of RAM accesses
  - It remains just 1 access in RAM (sometimes 2)!
  - We cut the grid in different blocks (with a vertical separation here)

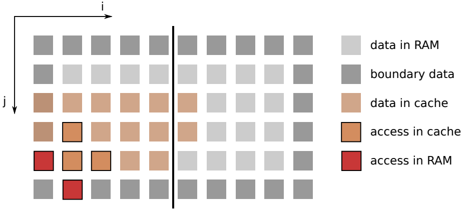
Cache blocking technique: example



Logical 2D grid memory view

- With the cache blocking technique we reduce the number of RAM accesses
  - It remains just 1 access in RAM (sometimes 2)!
  - We cut the grid in different blocks (with a vertical separation here)

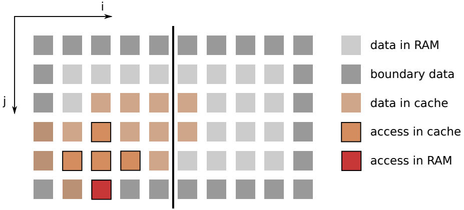
Cache blocking technique: example



Logical 2D grid memory view

- With the cache blocking technique we reduce the number of RAM accesses
  - It remains just 1 access in RAM (sometimes 2)!
  - We cut the grid in different blocks (with a vertical separation here)

Cache blocking technique: example



Logical 2D grid memory view

- With the cache blocking technique we reduce the number of RAM accesses
  - It remains just 1 access in RAM (sometimes 2)!
  - We cut the grid in different blocks (with a vertical separation here)

Cache blocking technique

- How to define the size of blocks?
  - It depends on the problem
  - In previous stencil code the optimal block size can be computed like this:

$$blockSize = \frac{sizeOfCache}{2 \times 3 \times nThreads \times sizeOfData},$$

with $sizeOfCache$ the size of the biggest cache (L3) in bytes, $nThreads$ the number of threads we are using during the code execution and $sizeOfData$ the size of data we are computing (simple precision = 4 bytes, double precision = 8 bytes).

- We divide by 2 because the caches are optimal when we use half of them
- We divide by 3 because we have to keep 3 rows in cache with the stencil problem
- Be careful, if $blockSize \geq cols$ then cache blocking is useless

Cache blocking technique: implementation

```
#define SIZE_OF_CACHE_L3 96 // bytes

void main()
{
  // pre-processing part
  int cols = 10, rows = 6;
  float *A = new float[cols*rows]; // in
  float *B = new float[cols*rows]; // out
  randomInit(A, cols*rows);
  int blockSize = SIZE_OF_CACHE_L3 / (2 * 3 * 1 * sizeof(float)); // (96 / 24) = 4

  // solver or computational part
  for(int iOff = 1; iOff < cols -1; iOff += blockSize) { // vertical blocks loop
    blockSize = min(cols -1 - iOff, blockSize); // reduce the block size if necessary
    for(int j = 1; j < rows -1; j++) // row
      for(int i = iOff; i < iOff + blockSize; i++) // column
        B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] +
                        A[(i   ) + (j   )*cols] +
                        A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];
  }

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Stencil code with cache blocking

Function calls

- A function call has a cost (extra assembly code)
- Is this is a sufficient reason to do not use functions a code?
  - It depends but sometimes yes it is!
  - In fact it depends on how many times we repeat the function call

```
void stencil(const float *A, float *B, const int i, const int j, const int cols)
{
  B[i + j*cols] = A[(i -1) + (j   )*cols] + A[(i +1) + (j   )*cols] +
                  A[(i   ) + (j   )*cols] +
                  A[(i   ) + (j -1)*cols] + A[(i   ) + (j +1)*cols];
}

void main()
{
  // pre-processing part
  int cols = 10, rows = 6;
  float *A = new float[cols*rows]; // in
  float *B = new float[cols*rows]; // out
  randomInit(A, cols*rows);

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      stencil(A, B, i, j, cols); // we call the stencil function many times!

  // post-processing part
  delete[] A;
  delete[] B;
}
```

Stencil code with function calls

What is inlining?

- Inlining a function is the same as replacing the function call by the code of the function itself
    - This way there is no more overhead because there are no more function calls
- We can manually do that but this is not a good idea...

How to perform inlining?

- In term of software engineering, functions (or methods in object-oriented programming) are well spread
  - It is much better to use functions for the code readability
- So, can we build a beautiful and efficient code?
  - This is language or compiler dependent...
  - In `C++` we have the `inline` keyword to perform inlining
  - In Fortran 90 (`ifort`) there is a directive: `!DEC$ ATTRIBUTES FORCEINLINE`
  - Often the compiler is free to perform inlining itself

```
inline void stencil(const float *A, float *B, const int i, const int j, const int cols) { ... }

void main()
{
  // pre-processing part ...

  // solver or computational part
  for(int j = 1; j < rows -1; j++) // row
    for(int i = 1; i < cols -1; i++) // column
      stencil(A, B, i, j, cols); // we call the stencil function many times
                                 // but with the inline keyword the compiler will
                                 // automatically replace the call by the inner code

  // post-processing part ...
}
```

Stencil code with function inlined calls in `C++`

Work with the compiler

- Today compilers provide a lot of options to auto apply optimizations or to improve the performance of codes
- In this lesson we will talk about the `C/C++` GNU compiler (`gcc`, `g++`) but you will find equivalent options with other compilers like the Intel compiler
- It is very important to know and understand what can and cannot be done by the compiler!
    - This way we can write beautiful (alias readable) and efficient codes
    - And the compiler can perform dirty optimizations when it generates the assembly code!

Optimize options with GNU compiler

- The most famous option `-O[level]`:
  - `-O0`: reduces compilation time and makes debugging produce the expected results, this is the default.
  - `-O1`: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
  - `-O2`: optimizes even more, GCC performs nearly all supported optimizations that do not involve a space-speed trade-off.
  - `-O3`: optimizes even more, turns on the `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-loop-vectorize`, `-ftree-loop-distribute-patterns`, `-ftree-slp-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` and `-fipa-cp-clone` options.
  - `-Ofast`: disregard strict standards compliance. It enables optimizations that are not valid for all standard-compliant programs. It turns on `-ffast-math` and the Fortran-specific `-fno-protect-parens` and `-fstack-arrays`.

Specific optimize options with GNU compiler

- Most of the time we will avoid to use specific options
- But it is important to understand what can be performed by some of them:
  - `-finline-functions`: activate automatic inlining, the compiler is free to perform or not the optimization.
  - `-ftree-vectorize`: activate auto-vectorization of the code.
  - `-ffast-math`: do not respect IEEE specifications for the calculations (we lose some precision) but it can severely improve performances.
  - `-funroll-loops`: unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. This option makes code larger, and may or may not make it run faster.
  - `-march=native`: allow specific instructions for a specific architecture, most of the time we will use this option in order to apply adapted vectorization on the code.
- The documentation of GNU optimize options: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Contents

1 Scalar optimizations

2 In-core parallelism
- Instruction-level parallelism
- Vectorization

3 Multi-core optimizations

Break instructions dependences

- Today CPUs need some independences between instructions
  - To fully use pipeline mechanism
  - And to efficiently exploit instruction-level parallelism (ILP)

```
void kernel(float *A, float *B, float *C, float *D, const float alpha, const int n)
{
  for(int i = 0; i < n; i++) {
    C[i] = A[i] + B[i];  // no dependences
    D[i] = C[i] * alpha; // D depends on C
    A[i] = D[i] - C[i];  // A depends on C and D
    B[i] = A[i] * 2.0f;  // B depends on A
  }
}
```

Kernel with a lot of dependences

Break instructions dependences

```c
void kernel(float *A, float *B, float *C, float *D, const float alpha, const int n)
{
  for(int i = 0; i < n; i += 2) {
    C[i+0] = A[i+0] + B[i+0]; // no dependences
    C[i+1] = A[i+1] + B[i+1]; // no dependences

    D[i+0] = C[i+0] * alpha;  // D[i+0] depends on C[i+0]
    D[i+1] = C[i+1] * alpha;  // D[i+1] depends on C[i+1]

    A[i+0] = D[i+0] - C[i+0]; // A[i+0] depends on C[i+0] and D[i+0]
    A[i+1] = D[i+1] - C[i+1]; // A[i+1] depends on C[i+1] and D[i+1]

    B[i+0] = A[i+0] * 2.0f;   // B[i+0] depends on A[i+0]
    B[i+1] = A[i+1] * 2.0f;   // B[i+1] depends on A[i+1]
  }
}
```

Kernel with less dependences

- Here, in the kernel example, it is possible to execute two instructions at the same time
- Sometimes compilers are capable of doing that on their own
- But when the code becomes complex the compiler can miss this kind of things

Break instructions dependences

```
#define UNROLL 2

void kernel(float *A, float *B, float *C, float *D, const float alpha, const int n)
{
  for(int i = 0; i < n; i += UNROLL) {
    for(int j = 0; j < UNROLL; j++)
      C[i+j] = A[i+j] + B[i+j]; // no dependences
    for(int j = 0; j < UNROLL; j++)
      D[i+j] = C[i+j] * alpha;  // D depends on C
    for(int j = 0; j < UNROLL; j++)
      A[i+j] = D[i+j] - C[i+j]; // A depends on C and D
    for(int j = 0; j < UNROLL; j++)
      B[i+j] = A[i+j] * 2.0f;   // B depends on A
  }
}
```

Kernel with less dependences ans less repetitions

- Previous code was quite ugly
- With the new one we can use the `-funroll-loops` compiler option (`gcc`) in order to unroll inner loops automatically
- And there are no more code repetitions!

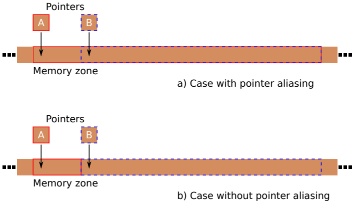
Code vectorization

- Today vector instructions provide an efficient way to improve code performances
- How can we use those instructions?
  - By enabling the auto-vectorization process: the compiler automatically detects the zones to vectorize and generates an assembly code with vector instructions (be sure to have `-ftree-vectorize` option activated)
  - By calling specific functions: intrinsics
  - By directly writing assembly code: we will try to avoid this solution!

Auto-vectorization

- Compilers are more and more capable of vectorizing codes automatically
- Loops are the best candidates for auto-vectorization
- This process requires respecting some constraints, the main idea is to write simple kernel codes (simple for the compiler)
    - The loop sizes have to be countable (`for-loops` without any `break`)
    - In general, branch statements (`if`, `switch`) are an obstacle to the vectorization
    - We have to guarantee that there is no pointer aliasing with the `__restrict` qualifier (this problem does not exist in `Fortran`)
        - There is pointer aliasing when two or more pointers can access the same memory zone
    - To achieve maximal bandwidth: loads and stores have to be aligned on the vector size

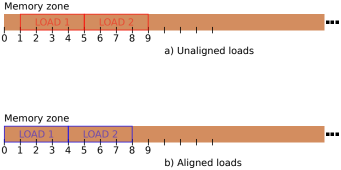
Pointer aliasing and aligned loads/stores



a) Case with pointer aliasing

b) Case without pointer aliasing

Aliasing problem illustration



a) Unaligned loads

b) Aligned loads

Alignment problem illustration

- a) A can access full B zone and B can access a sub-part of A
- b) A and B cannot cross

- a) Load 1 and 2 are not aligned on a multiple of 4
- b) Load 1 and 2 are well aligned on a multiple of 4

How to verify if the code has been well vectorized?

- Most simple way is to take a look at the restitution time
- But sometimes we cannot assert things with the time alone
- GNU compilers provide vectorization reporting with the following option: `-ftree-vectorizer-verbose=[level]`
  - `0`: tells the compiler to report no diagnostic information.
  - `1`: tells the compiler to report on vectorized loops.
  - `2`: tells the compiler to report on vectorized and non-vectorized loops.
  - `3`: tells the compiler to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
  - `4`: tells the compiler to report on non-vectorized loops.
  - `5`: tells the compiler to report on non-vectorized loops and the reason why they were not vectorized.

Intrinsic calls

- Sometimes the compiler does not succeed in automatically vectorizing the code
- In this case we have to be more explicit and use intrinsic functions
- Basically an intrinsic call is equivalent to an assembly instruction
- This type of functions are very very hardware dependent!
  - This is why we will try not to use them unless we do not have the choice
- x86 intrinsics documentation: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

Intrinsic calls: example

```c
// __restrict qualifier specify the compiler
// that there is no aliasing
void addVectors(const float* __restrict A,
                const float* __restrict B,
                      float* __restrict C,
                const int n)
{
  for(int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
}
```

Simple `addVectors` implementation

Intrinsic calls: example

```c
// __restrict qualifier specify the compiler
// that there is no aliasing
void addVectors(const float* __restrict A,
                const float* __restrict B,
                      float* __restrict C,
                const int n)
{
  for(int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
}
```

Simple `addVectors` implementation

- As you can see the intrinsics version is much more complex than the traditional version
- We have to limit intrinsics utilization for code readability

```c
// headers for intrinsic AVX functions
#include "immintrin.h"

void iAddVectors(const float* __restrict A,
                 const float* __restrict B,
                       float* __restrict C,
                 const int n)
{
  // with AVX-256 we can compute vector of
  // size 8 in single precision
  for(int i = 0; i < n; i += 8) {
    // load memory into vector registers
    __m256 rA = _mm256_load_ps(A +i);
    __m256 rB = _mm256_load_ps(B +i);

    // perform SIMD/vectorized addition
    __m256 rC = _mm256_add_ps(rA, rB);

    // store C vector register into memory
    _mm256_store_ps(C +i, rC);
  }
}
```

Intrinsics AVX-256 `iAddVectors` implementation

Contents

1 Scalar optimizations

2 In-core parallelism

3 Multi-core optimizations

- OpenMP reminders
- Avoid false sharing
- Reduce threads synchronisations
- Search algorithms

Multi-core codes

- Multi-core architecture is well spread in the High Performance Computing
- There are two main ways to use multi-core architecture
  - Create multiple processes with MPI as a standard (distributed memory model)
  - Or create multiple threads with OpenMP as a standard (shared memory model)
- In this lesson we will not speak about multiple processes model
- And we will go deeper into the multi-threaded model

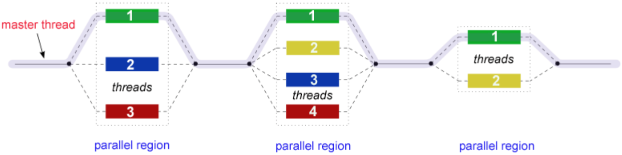
OpenMP

- OpenMP is a specific language for creating multi-threaded codes
- It is based on directives
  - Those directives describe how to perform the parallelism
  - The main advantage of directives is to not modify sequential code (in theory...)

```c
void addVectors(const float* __restrict A,
                const float* __restrict B,
                      float* __restrict C,
                const int n)
{
#pragma omp parallel // creation of a parallel zone directive (threads creation)
{
#pragma omp for // for-loop indices distribution directive
  for(int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
}
}
```

Simple `addVectors` OpenMP implementation

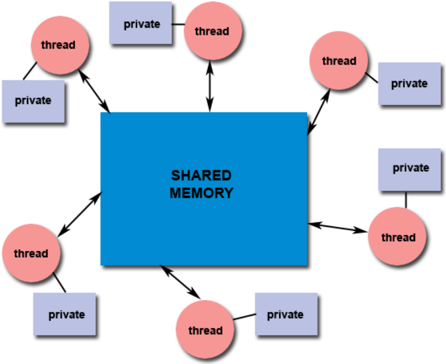
OpenMP: fork-join model



Fork-join model illustration (Wikipedia)

- OpenMP follows the fork-join model
  - Each time we create a parallel zone (`#pragma omp parallel`) we create threads (fork operation)
  - At the end of a parallel zone threads are destroyed and there is an implicit barrier (join operation)
    - Of course master thread remains

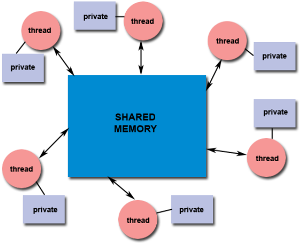
OpenMP: shared memory model



Shared memory model illustration (computing.llnl.gov)

- OpenMP also follows the shared memory model
  - Each thread can access a global memory zone: the shared memory
  - But threads own also private data (not completely shared model)

OpenMP: shared memory model example



Shared memory model illustration

```
void addVectors(const float* __restrict A,
                const float* __restrict B,
                      float* __restrict C,
                const int n)
{
#pragma omp parallel
{
#pragma omp for
  // i is private because it is declared
  // after the omp parallel directive
  for(int i = 0; i < n; i++)
    // A, B and C are shared!
    C[i] = A[i] + B[i];
}
}
```

Simple `addVectors` OpenMP implementation

OpenMP: control data range

- OpenMP provides data range control
  - `private`: local to the thread,
  - `firstprivate`: local to the thread and initialized
  - `shared`: shared by all the threads, in `C/C++` this is the default
- Here `alpha` is a constant, we can put it in the private memory of each thread
- An efficient parallelism comes with minimal synchronisations
  - Shared data can generate a lot of synchronisations
  - Privacy increases thread independence

```
void dot(const float* __restrict A,
               float* __restrict B,
         const float alpha,
         const int n)
{
#pragma omp parallel     \
            shared(A, B) \
            firstprivate(alpha, n)
{
#pragma omp for
  for(int i = 0; i < n; i++)
    B[i] = alpha * A[i];
}
}
```

OMP data range example

OpenMP: for-loop indices distribution

- For-loop indices distribution can be controlled by the `schedule` clause
  - `static`: indices distribution is precomputed, and the amount of indices is the same for each thread
  - `dynamic`: indices distribution is done in real time along the loop execution, work load balancing can be better than with the `static` scheduling but `dynamic` scheduling costs some additional resources in order to attribute indices in real time
- In HPC, `static` distribution is the best choice if we are sure that each iteration has the same cost (in time)
- There are other types of scheduling but this is not a full OpenMP lesson

```
// ...
#pragma omp for schedule(static, 128) //we statistically attribute 128 per 128 indices to each threads
  for(int i = 0; i < n; i++)
    B[i] = alpha * A[i];
}
// ...
```

OMP scheduling example

OpenMP: go further

- Previous slides were a brief overview of the main OpenMP principles
- To have more precise informations you can take a look at the very good OpenMP reference card:
  `http://openmp.org/mp-documents/OpenMP-4.0-C.pdf`
  - It could be a very good idea to print it and keep it ;-)
- In the next slides we will pay attention to OpenMP codes optimizations

Avoid false sharing

- False sharing is a phenomena that occurs when threads write simultaneously data in a same line
  - Remember, the cache system works on lines of words: a line is the smallest element in caches coherence mechanism
    - If two or more threads are working on the same line they cannot write data simultaneously!
    - Stores are serialized and we talk about false sharing
- To avoid false sharing, threads have to work on a bigger amount of data than the cache line size
  - Concretely we have to avoid `(static,1)` or `(dynamic,1)` scheduling
  - Cache lines are not very big ($\approx$ 64 Bytes)
  - Just putting a `(static,16)` or `(dynamic,16)` resolves the problem
  - Be aware that default OpenMP scheduling is `(static,1)`!

Reduce threads synchronisations: barriers

- In OpenMP there are a lot of implicit barriers, after each
  - `#pragma omp parallel` directive
  - `#pragma omp for` directive
  - `#pragma omp single` directive
- But not after `#pragma omp master` directive!
- If we are sure that there is no need to synchronise threads after the `#pragma omp for` directive, we can use the `nowait` clause
- Optimally we need only one `#pragma omp parallel` directive in a fully parallel code
  - OpenMP manages a pool of threads in order to reduce the cost of the `#pragma omp parallel` directive but this is not free, each time OpenMP has to reorganize the pool and wakes up the required threads

Reduce threads synchronisations: barriers

```
void kernelV1(const float *A, // size n
              const float *B, // size n
              const float *C, // size n
                    float *D, // size 2n
              const float alpha,
              const int n)
{
// threads creation overhead and
// private variables creation overhead
#pragma omp parallel shared(A, B, D) \
                     firstprivate(alpha, n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++)
    D[i] = alpha * A[i] + B[i];
} // implicit barrier
} // implicit barrier

// threads attribution overhead and
// private variables creation overhead
#pragma omp parallel shared(A, C, D) \
                     firstprivate(n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++)
    D[n+i] = A[i] + C[i];
} // implicit barrier
} // implicit barrier
}
```

A lot of OMP barriers

Reduce threads synchronisations: barriers

```c
void kernelV1(const float *A, // size n
              const float *B, // size n
              const float *C, // size n
                    float *D, // size 2n
              const float alpha,
              const int n)
{
// threads creation overhead and
// private variables creation overhead
#pragma omp parallel shared(A, B, D) \
                     firstprivate(alpha, n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++)
    D[i] = alpha * A[i] + B[i];
} // implicit barrier
} // implicit barrier

// threads attribution overhead and
// private variables creation overhead
#pragma omp parallel shared(A, C, D) \
                     firstprivate(n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++)
    D[n+i] = A[i] + C[i];
} // implicit barrier
} // implicit barrier
}
```

A lot of OMP barriers

```c
void kernelV2(const float *A, // size n
              const float *B, // size n
              const float *C, // size n
                    float *D, // size 2n
              const float alpha,
              const int n)
{
// threads creation overhead and
// private variables creation overhead
#pragma omp parallel shared(A, B, C, D) \
                     firstprivate(alpha, n)
{
#pragma omp for schedule(static,16) nowait
{
  for(int i = 0; i < n; i++)
    D[i] = alpha * A[i] + B[i];
} // no implicit barrier (nowait clause)

#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++)
    D[n+i] = A[i] + C[i];
} // implicit barrier
} // implicit barrier
}
```

Less OMP barriers

Reduce threads synchronisations: critical sections

- Sometimes it is not possible to have a fully parallel code and some regions of the code remain intrinsically sequential
- In OpenMP we can specify this kind of region with the `#pragma omp critical` directive
- But we have to use this directive carefully
  - It can be a main cause of slow down in OpenMP codes!

Reduce threads synchronisations: critical sections

```
float kernelV1(const float *A, // size n
                     float *B, // size n
               const int n)
{
  float minVal = INF;

#pragma omp parallel shared(A, B, minVal) \
                     firstprivate(n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    B[i] = 0.5f * A[i];

#pragma omp critical // we are sure that only
                     // one thread can
                     // modify minVal
{
    if(B[i] < minVal)
      minVal = B[i];
}
  }
}
}

  return minVal;
}
```

Critical section

Reduce threads synchronisations: critical sections

```
float kernelV1(const float *A, // size n
                     float *B, // size n
               const int n)
{
  float minVal = INF;

#pragma omp parallel shared(A, B, minVal) \
                     firstprivate(n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    B[i] = 0.5f * A[i];

#pragma omp critical // we are sure that only
                     // one thread can
                     // modify minVal
{
    if(B[i] < minVal)
      minVal = B[i];
}
  }
}
}

  return minVal;
}
```

Critical section

```
float kernelV2(const float *A, // size n
                     float *B, // size n
               const int n)
{
  float minVal = INF;

#pragma omp parallel shared(A, B, minVal) \
                     firstprivate(n)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    B[i] = 0.5f * A[i];

    // when B[i] >= minVal there is no more
    // threads synchronisation
    if(B[i] < minVal)
#pragma omp critical
{
      // this is very important to re-do the
      // test because an other thread may
      // have modify the minVal value
      if(B[i] < minVal)
        minVal = B[i];
}
  }
}
}

  return minVal;
}
```

Critical section optimized

Search algorithms

- In OpenMP 3 there is no optimal solution for search algorithms
- This kind of algorithm typically requires while-loops or do-while-loops
- However there is a tip to fix this lack in OpenMP 3
- Latest version of OpenMP (v4) provides better control of threads
  - We can terminate threads...
  - We will not speak about OpenMP 4 because many current systems does not support this version

Search algorithms: OpenMP 3 tip

```
bool searchValV1(const float *A,
                 const int n,
                       float val)
{
  bool found = false;

#pragma omp parallel shared(A, found) \
                     firstprivate(val)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    if(A[i] == val)
      found = true;
  }
}
}

  return found;
}
```

Search algorithms

Search algorithms: OpenMP 3 tip

```
bool searchValV1(const float *A,
                 const int n,
                       float val)
{
  bool found = false;

#pragma omp parallel shared(A, found) \
                     firstprivate(val)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    if(A[i] == val)
      found = true;
  }
}
}

  return found;
}
```

Search algorithms

```
bool searchValV2(const float *A,
                 const int n,
                       float val)
{
  bool found = false;

#pragma omp parallel shared(A, found) \
                     firstprivate(val)
{
#pragma omp for schedule(static,16)
{
  for(int i = 0; i < n; i++) {
    if(!found) // we are doing nothing if
               // we have found the value
               // in the array
      if(A[i] == val)
        found = true;
  }
}
}

  return found;
}
```

Search algorithms optimized

Final words

- Of course there are many more possible optimizations
- The purpose of this lesson was to raise awareness among optimization problematic
- Now you have tools to understand and to create your own optimizations
- Presented techniques are very often specific to the problem nature
  - Be aware that there is no perfect optimization
  - You will have to think about your needs before trying to optimize your code