Advanced Optimization Techniques

ICTP Trieste 2014

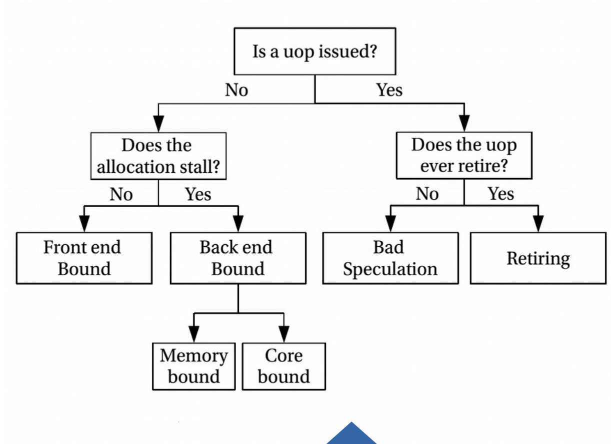
Dr. Christopher Dahnken

Intel GmbH

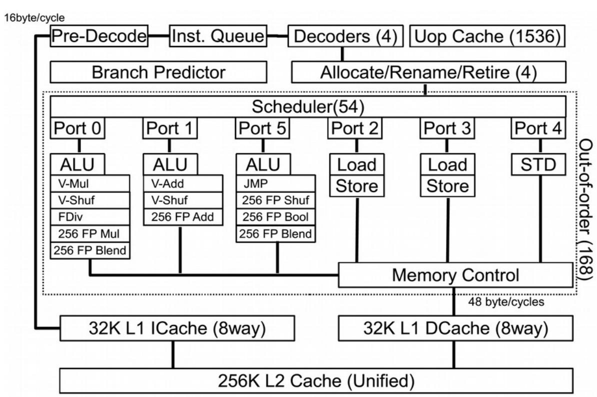
Legal Disclaimer & Optimization Notice

Outline

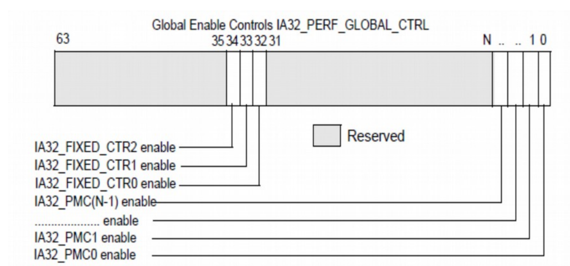
Method



CPU



Measurement



Code

```
!$OMP SECTION
!   tsend=dclock()
 if(iblock.lt.(nblocks)) then
    nxti=m_of_i(iblock+1)
    nxtj=n_of_i(iblock+1)
    nxtk=k_of_i(iblock+1)

    nxt_buffsize_m=buffersize(ms,bm,nxti)
    nxt_index_m=(nxti-1)*bm+1

    nxt_buffsize_n=buffersize(ns,bn,nxtj)
    nxt_index_n=(nxtj-1)*bn+1

    nxt_buffsize_k=buffersize(ks,bk,nxtk)
    nxt_index_k=(nxtk-1)*bk+1
```

Using the Intel compiler

Dealing with common problems

- In the following, we will discuss the most common issues for sub-optimal performance
- Compute bound code : pipeline optimization and vectorization
- Memory bound kernels : cache and memory optimization
- Branchy kernels : overcoming branch penalties and exploiting the branch predictor for performance

Dealing with Branches
(some of the features here have vectorization implications as well)

Dealing with branches

- Reminder: Branches are points in the code where the instruction pointer is set to another address, either conditionally or unconditionally.
- The branch prediction unit predicts branches based on previous behavior. Wrongly predicted branches lead to a pipeline flush, resulting penalty cycles proportional to the pipeline length.

Dealing with branches builtin_expect

builtin_expect

__builtin_expect is a compiler intrinsic that points the compiler to which branch criterion will likely occur:

```
if(__builtin_expect(x<0,1)){
    somefunction(x);
} else{
    someotherfunction(x);
}
```

Dealing with branches
Inlining

Inlining function calls

- A call of a function translates into a unconditional branch in the assembly instructions
- If the code content of the function is small, this may result in a performance penalty
- For instance,
  ```
  int inc_by_one(int i){
          return i+1;
  }
  ```
  is probably not a brilliant idea :-)
- Also, inlining enables better optimization, since the function content can be inspected by the compiler in the execution context.

Dealing with branches
Inlining

- **Inlining the function definition with C/C++ keyword "inline"**

```
inline int inc_by_one(int i){
    return i+1;
}
```

This will affect all calls of the function

- **Inlining at the function call with a compiler pragma**

```
#pragma inline [recursive]
#pragma forceinline [recursive]
#pragma noinline
```

Example:

```
#pragma forceinline
j=inc_by_one(j);
```

Dealing with branches
Inlining

Caveat

Excessive (recursive) inlining might blow up compile times and required memory

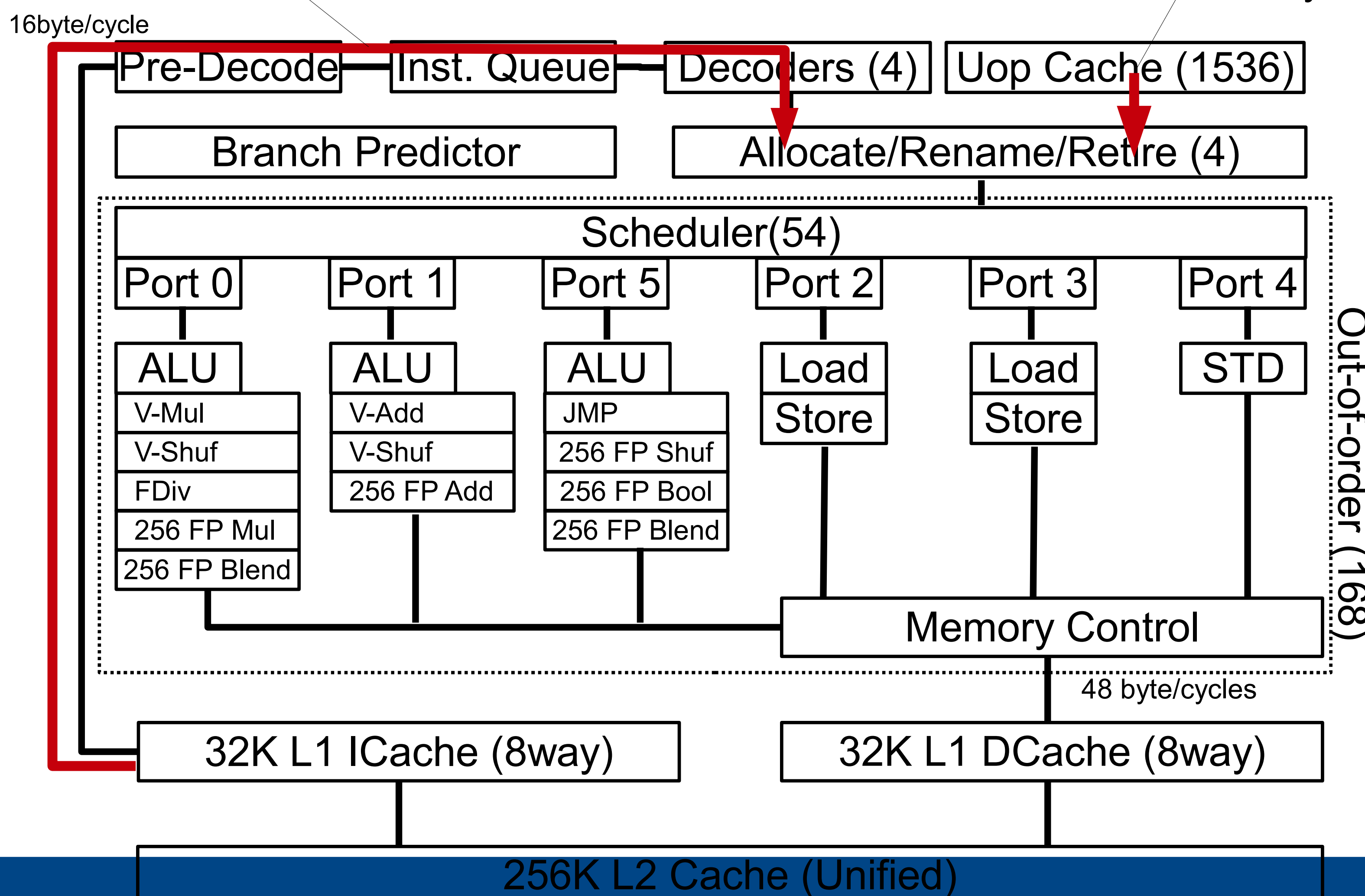
Excessive (recursive) inlining might cause the code size to blow up and in consequence generate front-end issues.

Let's look at the core block diagram ...

Dealing with branches
Inlining

Long, non-branchy codes are more likely to come this way

Compact, branchy codes are more likely to come this way



Dealing with branches
Profile Guided Optimization

Profile Guided Optimization (PGO)

PGO is a three step process particularly suited to eliminate branch codes

- Create an instrumented binary with the compiler option -prof-gen.
- Run this binary with one or more representative workloads. This will create profile files containing the desired information.
- Compile once more with the compiler option -prof-use.

The compiler will then optimize branch statement.

Dealing with branches
Unrolling

Each loop (usually) consists of a loop variable, a comparison and a condition jump back to the beginning of the loop body

```
int s=0;
for(int i=0;i<16;i++){s+=i;}
```

Translates:

```
400545:    8b 45 fc       mov    -0x4(%rbp),%eax
400548:    01 45 f8       add    %eax,-0x8(%rbp)
40054b:    83 45 fc 01    addl   $0x1,-0x4(%rbp)
40054f:    83 7d fc 0f    cmpl   $0xf,-0x4(%rbp)
400553:    7e f0          jle    400545 <main+0x18>
```

Dealing with branches
Unrolling

- Most of the loops body actually consists of code maintaining the loop itself (3 of 5 instruction) vs the actual workload (2 of 5)
- Since we know the length of the loop, we could unroll it 2x, 4x, … or even fully!
- One can do that manually (of course) or let the compiler take care for it ...

Dealing with branches
Unrolling

Unrolling with compiler pragmas

```
#pragma unroll
#pragma unroll(n)
#pragma nounroll
```

Placing “**`#pragma unroll (2)`**” in front of the loop unrolls the loop twice. The compiler takes care of the code for remainders (when looplength % unrollfactor !=0).

Dealing with branches
Unrolling

<u>Unrolling with compiler pragmas</u>

```
int s=0;
#pragma unroll (2)
for(int i=0;i<16;i++){s+=i;}
```

Is equivalent to

```
for(int i=0;i<16;i+=2){s+=i;s+=i+1;}
```

Dealing with branches
Unrolling

<u>Caveat</u>

- As with inlining, excessive unrolling can hit the performance rather than help (same reasons)
- The CPU has a so-called Loop Stream Detector (LSD). This piece of hardware allows small loops (~28 instructions) to be executed very quickly – Generally, unrolling is preferable, though.

Dealing with branches
Unroll and Jam

When dealing with the unrolling of nested loops, you don't necessarily want the loop body repeated trivially, but cleverly combined into an inner loop body.

The pragma unroll_and_jam can do exactly this. Let's look at an example ...

Dealing with branches
Unroll and Jam

```
for(int i=0;i<size;i++){
#pragma unroll(2)
  for(int j=0;j<size;j++){
    for(int k=0;k<size;k++){
      c[i*size+j]
        +=a[i*size+k]*b[k*size+j];
    }
  }
}
```

This results in the following ...

Dealing with branches
Unroll and Jam

```
for(int i=0;i<size;i++){
  for(int j=0;j<size;j+=2){
    for(int k=0;k<size;k++){
      c[i*size+j]
        +=a[i*size+k]*b[k*size+j];
    }
    for(int k=0;k<size;k++){
      c[i*size+j+1]
        +=a[i*size+k]*b[k*size+j+1];
    }
  }
}
```

Unrolling results in the two inner loop bodies are simply replicated. But ...

Dealing with branches
Unroll and Jam

```
for(int i=0;i<size;i++){
#pragma unroll_and_jam(2)
  for(int j=0;j<size;j++){
    for(int k=0;k<size;k++){
      c[i*size+j]
        +=a[i*size+k]*b[k*size+j];
    }
  }
}
```

… gives a result equivalent to the following ...

Dealing with branches
Unroll and Jam

```
for(int i=0;i<size;i++){
  for(int j=0;j<size;j+=2){
    for(int k=0;k<size;k++){
      c[i*size+j]
        +=a[i*size+k]*b[k*size+j];
      c[i*size+j+1]
        +=a[i*size+k]*b[k*size+j+1];
    }
  }
}
```

Much better! The compiler is also able to reuse this entry!

Dealing with branches
Unroll and Jam

Unroll and Jam

```
#pragma unroll_and_jam
#pragma unroll_and_jam (n)
#pragma nounroll_and_jam
```

Remarks:

Only when -O3 is used! This is a bit surprising, but the compiler documentation claims so ...

Dealing with branches Exploiting the branch predictor

- So far we have considered avoiding or eliminating branches
- The branch predictor might also fully work to ones advantage, however!
- Let's see how we can use always true if-conditions to generate highly optimized code ...

Dealing with branches
Exploiting the branch predictor

Consider the generic computation of a polynomial:

```
double mypolynomial(c,x,degree){
  double ret=0;
  for(int i=0;i<degree; i++){
    ret+=c[i]*pow(x,i);
  }
  return ret;
}
```

$$p(x)=\sum_i c_i x^i$$

Dealing with branches
Exploiting the branch predictor

- In many cases, one attempts a single degree throughout a whole run, say “approximation to the 8th degree” or “second order perturbation theory”.
- If you know that you need a specific configuration most of the time, the branch predictor works to you advantage since it will predict correctly for the overwhelming part ...

Dealing with branches
Exploiting the branch predictor

The compiler optimizes this much easier than this.

```
double mypolynomial(double* c,
          double* x, int degree){
  if(degree==4){
    ret=c[0]+c[1]*x+c[2]*x*x+c[3]*x*x*x;
  }else{
    ret=mygeneralpolynomial(c,x,degree)
  }
  return ret;
}
```

The BPU will always predict correctly when you use degree=4 throughout the run.

Dealing with vectorization

Why doesn't my code vectorize?

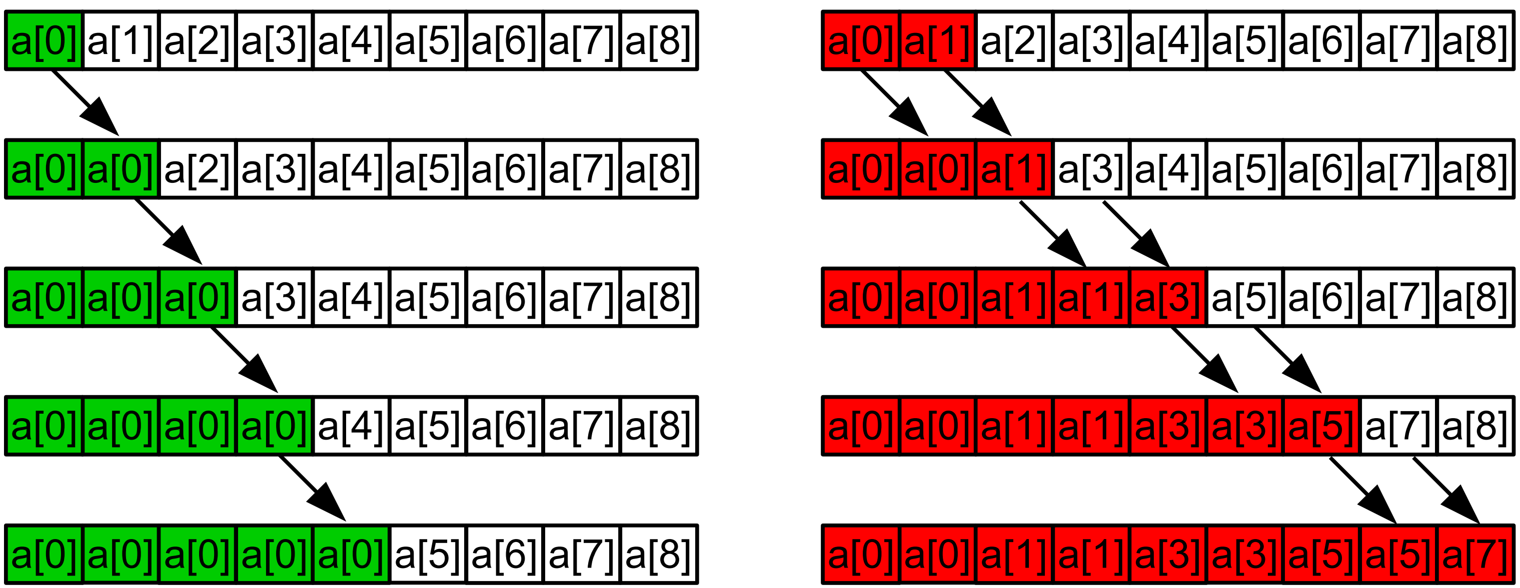
Most important reasons why the compiler won't vectorize you code:

- (Vector) Data Dependences
- Data Aliasing
- Too Complex
- Not efficient

Vectorization
Vector Dependences

```
for(int i=0;i<length-1;i++){
        a[i+1]=a[i];
}
```

Sequential | **2-element vector**



Rightfully, the compiler won't vectorize!

Vectorization Aliasing

```
void mulshift(double* a, double* b, double* c){
   for(int i=0;i<length;i++){
      c[i+1]=a[i]*b[i];
    }
}
```

- The compiler cannot prove that, say, c and a are not pointing to the same array
- It might assume that a and c are the same and a vector dependence exists (like in the previous example).

Vectorization Inefficiency

- In some cases the compiler claims that it guesses that the vectorization would be not efficient, so was avoided. This can, for example, happen when accessing non-contiguous memory.
- Example:

```
for(int i=0;i<length;i++){
   sum+=a[i+offset];
}
```

This would need a so-called gather, which is likely really not efficient for such a short computation.

Vectorization
Aliasing

Compiler flags

- (-no)-ansi-alias: enables the ANSI rules for aliasing (reads: the programmer is responsible for checking that passed arrays don't alias). The default is -ansi-alias, so aliasing shouldn't often avoid vectorization.
- -f(no)-alias: aliasing will not be assumed in the file that is compiled using this option – use carefully. Default -falias.

Vectorization
Aliasing

Using the restrict keyword

restrict: Hints the compiler that the restricted pointers do not alias. Does only work with -std=c99. Does not work with C++.

```
void mulshift(double* restrict a, double*
restrict b, double* restrict c){
for(int i=0;i<length;i++){
     c[i+1]=a[i]*b[i];
   }
}
```

This will vectorize even if -no-ansi-alias is defined!

Vectorization
Compiler directives

- The compiler offers three pragmas that have different impact on the vectorizer:
- **`#pragma ivdep`**
- **`#pragma vector`**
- **`#pragma simd`**
- All pragmas go in front of the loop you want to vectorize. We will discuss them one by one ...

Vectorization
Compiler directives

- **`#pragma ivdep`**
- Tells the compiler that assumed vector dependeces in the following loop should be ignored. Proven vector dependeces are not affected!
- This pragma is available with most compilers, although the implementation might differ.

Vectorization
Compiler directives

- **`#pragma vector`**
- Similar in function as ivdep, but has additional optional clauses:
- **`#pragma vector always:`** Overrides the compiler heuristics
- **`#pragma vector [un]aligned:`** Tells the compiler to use (un)aligned data movement
- **`#pragma vector [non]temporal [vars]:`** Tells the compiler to use streaming stores in case of nontemporal, which writes the data directly to memory and doesn't pollute the cache. Takes a comma spearated list of variables that should be treated nontemporal.

Vectorization
Compiler directives

- **`#pragma simd`**
- This is the most powerful of the vector pragmas
- Tells the compiler to ignore any heuristics or dependence, proven or not
- The programmer is fully repsonsible for securing correctness
- Supports many optional clauses, some with function similar to the OpenMP **`parallel for`** pragma

Vectorization
Compiler directives

- **`#pragma simd vectorlength(length)`**
- Tells the compiler to use a specific vector length. The argument length must be a power of two. Idealy, this is the maximum length for the architecture and data type under consideration.

Example:

```
void foo(float* a,float* b,float* c)
#pragma simd vectorlength(8)
for(int i=0;i<length;i++)
   a[i]=b[i]*c[i];
}
```

An 256bit AVX vector can take 8 32bit floats

Vectorization
Compiler directives

- **`#pragma simd vectorlengthfor(datatype)`**
- Tells the compiler to choose the appropriate vector length for this data type. The argument length must be a data type, e.g. float, double, int, etc ....

<u>Example:</u>

```
void foo(float* a,float* b,float* c)
#pragma simd vectorlengthfor(float)
for(int i=0;i<length;i++)
   a[i]=b[i]*c[i];
}
```

Vectorization
Compiler directives

- **`#pragma simd private(var1,[var2,...])`**
- Tells the compiler that the variables var1 [, var2,...] are treated to be independent in each loop iteration. The initial and final values are undefined. **`firstprivate`** and **`lastprivate`** are also present, with similar functionality as in OpenMP.

Example:

```
#pragma simd private(c)
for(int i=0;i<length;i++)
   c=i;
   a[i]=c*b[i];
}
```

Vectorization
Compiler directives

- **`#pragma simd reduction(op:var)`**
- Tells the compiler perform a reduction with the specified operation **`op`**. After the loop, the variable **`var`** will hold the correct value of the reduction

<u>Example:</u>

```
#pragma simd reduction(+:c)
for(int i=0;i<length;i++)
    c+=a[i];
}
```

Vectorization
Array Notations

- Array Notations (AN) is Intel-specific language extension introduced with Cilk Plus
- AN allows the direct expression of data parallelism
- Relieves the compiler of the dependence and aliasing analysis (to a degree) and provides an easy way to write correct, performing code.

Vectorization
Array Notations

- AN introduces an array section notation that allows the specification of particular elements, compact or regularly strided:

```
<array base>[<lower bound>:<length>:<stride>]
```

- **Example**

```
a[:]        //the whole array
a[0:10]     //elements 0 through 9
a[0:5:2]   // elements 0,2,4,6,8
```

Vectorization
Array Notations

- **<u>More Examples</u>**:

```
// element-wise multiplication
c[0:10]=a[0:10]*b[0:10];
// increment all elements
a[0:10]++;
// m[i] will contain 1 if a[i]<b[i], 0
otherwise
m[0:10]=a[0:10]<b[0:10];
//works with multiple ranks
a[0:10][0:10]=b[10:10][10:10];
// or even from totally different ranks!
a[0:10][0:10]=b[10:10][2][10:10];
```

Summary

- The Intel compiler offers a plethora of switches and pragmas for dealing with branchy or non-vectorizing code
- Cilk Plus Array Notation is a portable and high level way of directly expressing data level parallelism
- … if you require even more control over AVX, you need to consider programming it directly, which we will discuss in the next module.