Dr. Pekka Manninen
CSC - IT Center for Science
Finland

Performance Optimization of Scientific Software

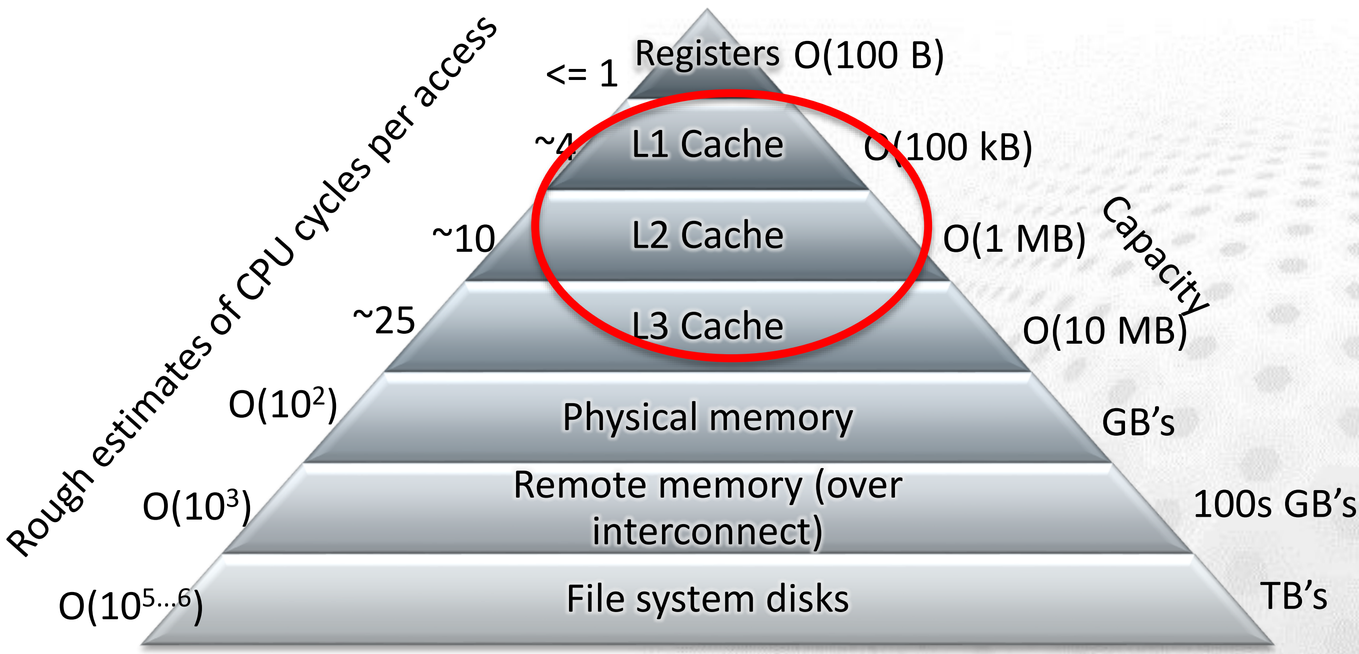
Part II: Node-Level Performance Tuning

CSC Webinar
November 6, 2018

Setting the scene

- Modern multicore CPUs are very complex (with evermore increasing complexity)
  - Multiple CPU cores within one socket
  - Superscalar out-of-order instruction execution with branch prediction
  - Multilevel coherent caches
  - SIMD vector units
  - SMT capabilities for multithreading
- Typical supercomputer node contains 2-4 sockets
- To get most out of the hardware, performance engineering is needed

Memory hierarchy

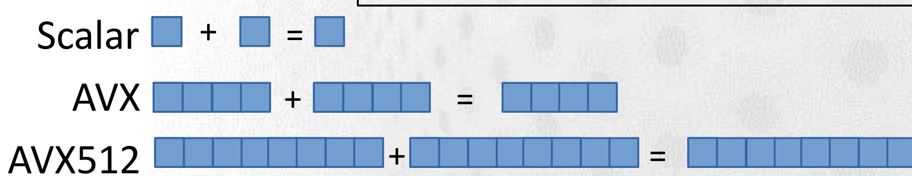


SIMD vectorization

- SIMD instructions operate on multiple elements at one cycle
- AVX/AVX2: 256 bits
  - 4 DP values or 8 SP values
  - Fused multiply-add (AVX2)
  - Haswell CPUs on Sisu
- AVX512: 512 bits
  - 8 DP values or 16 SP values
  - Current generation

```
double * A, * B, * C;
int i, N;

for (i=0; i<N; i++)
   C[i]=B[i]+A[i];
```



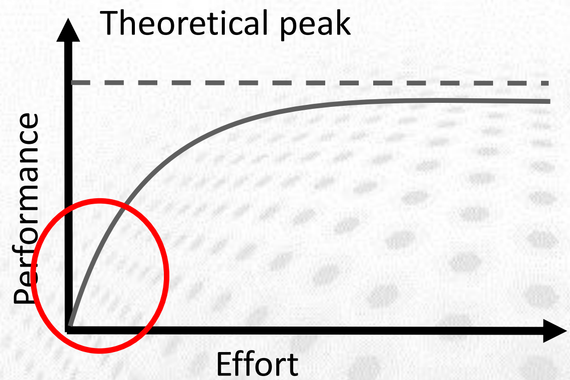
Recall: Finding single-core hotspots

- Signature: Low L1 and/or L2 cache hit ratios
  - <96% for L1, <99% for L1+L2
  - Issue: **Bad cache utilization**
- Signature: Low vector instruction usage
  - Issue: **Non-vectorizable (hotspot) loops**
- Signature: Traced ”math” group featuring a significant portion in the profile
  - Issue: **Expensive math operations**

SLIGHT DETOUR: OPTIMAL PORTING

Optimal porting

- ”Improving application performance without touching the source code”
  - Compilers & compiler flags
  - Numerical libraries
  - MPI rank placement
  - Thread affinities
  - Filesystem parameters



- Potential to get significant performance improvements with little effort
- Should be revisited routinely

Choosing a compiler

- Many different choices
  - GNU, PGI, Intel, Cray, XL etc.
- Compatibility
  - Different proprietary intrinsics
  - Different rounding rules
- Compilers tend to be cautious with optimization
- Performance
  - There is no universally fastest compiler
  - Depends on the application or even input

Compiler optimization techniques

- Architecture-specific tuning
  - Tunes all applicable parameters to the defined microarchitecture
- Vectorization
  - Exploiting the vector units of the CPU (AVX etc.)
  - Improves performance in most cases
- Loop transformations
  - Fusing, splitting, interchanging, unrolling etc.
  - Effectiveness varies

Compiler flag examples

Feature	Cray	Intel	GNU
Listing	-hlist=a	-qopt-report=3	-fopt-info-vec
Balanced Optimization	(default)	-O2	-O3
Aggressive Optimization	-O3 –hfp4	-Ofast	-Ofast –funroll-loops
Architecture specific tuning	-h cpu=<target>	-x<target>	-march=<target>
Fast math	-hfp4	-fp-model fast=2	-ffast-math
More info (on sisu.csc.fi)	man crayftn / man craycc	icc --help ifort --help	man gcc man gfortran

Compiler optimization techniques

- Compilers tend to be cautious with optimization - when compiling scientific software you can typically have an ”all-in” approach
- If something breaks down, find the routine that causes the trouble and compile that file with less aggressive optimization and the rest with the aggressive levels

Compiler feedback/output

- Compilers will be more verbose on what they are doing for you code when requested by a specific compiler flag
- Cray compiler: `ftn -rm … or cc/CC -hlist=m …`
  - Compiler generates an <source file name>.lst file that contains annotated listing of your source code
- Intel compiler: `ftn/cc -qopt-report=3 -vec-report=6`
  - See `ifort/icc --help reports`
- GNU compiler: `ftn/cc: -fopt-info-vec`

Doesn't the compiler do everything?

- You can make a big difference to code performance
  - Helping the compiler spot optimisation opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) “optimizations” in older code
    - Simple code is the best, until otherwise proven
- First, check what the compiler is already doing
- Use the performance analysis data to establish understanding on the performance bottlenecks & shortcomings

ADDRESSING BAD CACHE UTILIZATION

General considerations for improved cache utilization

- Always try to use all data in cache line (64 bytes)
  - Memory is always read in terms of cache lines
- Use regular access patterns
  - Helps hardware prefetchers
- Try to re-use data, so that data loaded into caches are used multiple times
  - Blocking of operations on high dimensional data
    - You can assist & control with compiler pragmas/directives
  - Sorting of data before operations
- Does structure-of-arrays (SoA) or array-of-structures (AoS) fit your work best?

Loop interchange

- If multi-dimensional arrays are addressed in a wrong (non-consecutive) order, it causes a lot of cache misses => horrible performance
  - C is row-major, Fortran column-major

```
do i=1,N
  do j=1,M
    sum = sum + a(i,j)
  end do
end do
```

```
do j=1,M
  do i=1,N
    sum = sum + a(i,j)
  end do
end do
```

  - The compiler may (but also may not) re-order loops automatically (see compiler diagnostics)

Loop fission/fusion

- Loop fission and fusion are optimization techniques to improve cache efficiency by improving the locality of reference to the variables within a loop
  - Loop *fission*: a large loop is divided into multiple loops
  - Loop *fusion*: multiple small loops are combined into a large loop
- When provided with sufficient information about the loop trip counts, the compiler automatically tries to perform loop fission/fusion based on performance heuristics

FIXING NON-VECTORIZATION OF LOOPS

General considerations for vectorization

- The compiler will only vectorize loops
- Unit strides are the best
- Indirect addressing will not vectorize (efficiently)
- Can vectorize across inlined functions but not if a procedure call is not inlined
- Needs to know loop tripcount (but only at runtime)
  - i.e. `while` style loops will not vectorize
- No recursion allowed

Helping the compiler

- Does the non-vectorized loop have true dependencies?
  - No: add the pragma/directive `ivdep` on top of the loop
    - Or the OpenMP SIMD pragma (**`#pragma omp simd`**)
    - C/C++: the **`__restrict__`** keyword for fixing aliasing
  - Yes: Accept the situation or try to rewrite the loop
- If you cannot vectorize the entire loop, consider splitting it - so as much of the loop is vectorized as possible

Example

- See compiler feedback on why some loops were not vectorized

```
127.  + 1------< for (i = 1; i < nx + 1; i++)
128.  + 1 r2---<   for (j = 1; j < ny + 1; j++) {
129.  + 1 r2         new[i][j] = old[i][j] + a * dt *
130.    1 r2           ((old[i+1][j] - 2.0 * old[i][j] + old[i-1][j]) / dx2 +
131.    1 r2            (old[i][j+1] - 2.0 * old[i][j] + old[i][j-1]) / dy2);
132.    1 r2-->>   }
```

```
CC-6290 CC: VECTOR File = heat.c, Line = 127
  A loop was not vectorized because a recurrence
was
  found between "old" and "new" at line 129.
CC-6308 CC: VECTOR File = heat.c, Line = 128
  A loop was not vectorized because the loop
  initialization would be too costly.
CC-6005 CC: SCALAR File = heat.c, Line = 128
  A loop was unrolled 2 times.
```

Runtime: 8.55 s

Example

Tell the compiler that old and new do not overlap

```
127.  + 1-------< for (i = 1; i < nx + 1; i++)
128.    1             #pragma ivdep
129.    1 Vr2---<     for (j = 1; j < ny + 1; j++) {
130.  + 1 Vr2           new[i][j] = old[i][j] + a * dt *
131.    1 Vr2              ((old[i+1][j] - 2.0 * old[i][j] + old[i-1][j]) / dx2 +
132.    1 Vr2               (old[i][j+1] - 2.0 * old[i][j] + old[i][j-1]) / dy2);
133.    1 Vr2-->>     }
```

```
CC-6294 CC: VECTOR File = ex7_heat.c, Line = 127
  A loop was not vectorized because a better candidate was
found at line 129.
CC-6005 CC: SCALAR File = ex7_heat.c, Line = 129
  A loop was unrolled 2 times.
CC-6204 CC: VECTOR File = ex7_heat.c, Line = 129
  A loop was vectorized.
```

Runtime: 6.55 s

REDUCING THE COST OF EXPENSIVE MATH OPERATIONS

General consideration

- The cost of different scalar floating-point operations is roughly as follows:
  - <= 1 cycle: +, *
  - ~20 cycles: /, `sqrt`, `1/sqrt`
  - ~100-300 cycles: `sin`, `cos`, `exp`, `log`, ...
- There is also instruction latency and secondary performance impact from issues related to the pipelining when using the most expensive operations

Strength reduction techniques

- Loop hoisting: try to get the expensive operations out of innermost loops
  - Precomputing values, look-up tables etc
- Consider replacing division (`a/b`) with multiplication by reciprocal (`a*(1/b)`)
  - Assuming you can compute 1/b less often than the original division itself
- Reduce the use of sin, cos, exp, log, pow by using identities, such as
  - `pow(x,2.5) = x*x*sqrt(x)`
  - `sin(x)*cos(x) = 0.5*sin(2*x)`
- Use vectorized versions of the operations (through library calls)

Part II take-home messages

- Do the performance analysis!
  - Then you know what to look for
- Utilize the compiler feedback
  - Check especially whether the hot-spot loops have been vectorized or not
  - Then you know the reason why some optimizations have not been applied, and you can assist the compiler to overcome those restrictions
- Utilize the CPU efficiently, especially caches and SIMD vector units
- Mind the way you implement your equations, the cost of arithmetic operations vary greatly

Optional lab

- From the lab instruction sheet available in the page of the first webinar, do now the sections 5 and 6
- The last part of the series will take place on November 20