BASIC OPTIMIZATION TECHNIQUES FOR SERIAL CODE

Dr Noor Mahammad Sk

Center for High Performance Reconfigurable Computing

Hprcse

Scalar Profiling

- Gathering information about a program's behavior, specifically its use of resources, is called *profiling.*
- The most important "resource" in terms of high performance computing is runtime.
- A common profiling strategy is to find out how much time is spent in the different functions, and lines of a code.
- This will help to identify *hot spots,*
- *The parts of the program that require the dominant* fraction of runtime.
- These hot spots are subsequently analyzed for possible optimization opportunities.

Identifying Bottlenecks in Performance

- Even if hot spots have been determined, it is sometimes not clear from the start what the actual reasons for a particular performance bottleneck
- In such a case whether data access to main memory or pipeline stalls limit performance.
- If data access is the problem, it may not be straightforward to identify which data items accessed in a complex piece of code actually cause the most delay.
- *Hardware performance counters may help to resolve such issues.*
- *They Provide* all current processors and allow deep insights into the use of resources within the chip and the system.
- When nothing can be done any more to accelerate a serial code any further.
- It is essential for the user to be able to identify the point where additional optimization efforts are useless.

Function- and Line-based runtime profiling

- Two technologies are used for function- and line-based profiling: Code *instrumentation and sampling*
- Instrumentation works by letting the compiler modify each function call, inserting some code that logs the call, its caller (or the complete call stack) and probably how much time it required.
- This technique incurs significant overhead, especially if the code contains many functions with short runtime.
- The instrumentation code will try to compensate for that, but there is always some residual uncertainty.

Function- and Line-based runtime profiling

- Sampling is less invasive:
- The program is interrupted at periodic intervals, e.g., 10 milliseconds, and the program counter (and possibly the current call stack) is recorded.
- Necessarily this process is statistical by nature, but the longer the code runs, the more accurate the results will be.
- If the compiler has equipped the object code with appropriate information, sampling can deliver execution time information down to the source line and even machine code level.
- Instrumentation is necessarily limited to functions or *basic blocks (code with one* entry and one exit point with no calls or jumps in between) for efficiency reasons.

Function Profiling

- The most widely used profiling tool is gprof from the GNU binutils package.
- gprof uses both instrumentation and sampling to collect a flat function profile as well as a callgraph profile, also called a *butterfly graph.*
- **In order to activate profiling:**
- The code must be compiled with an appropriate option (many modern compilers can generate gprof-compliant instrumentation; for the GCC, use -pg) and run once.
- This produces a non-human-readable file gmon.out, to be interpreted by the gprof program.

Flat Profile

- The flat profile contains information about execution times of all the program's functions and how often they were called:
- There is one line for each function.

1	%	cumulative	self		self	total	
2	time	seconds	seconds	calls	ms/call	ms/call	name
3	70.45	5.14	5.14	26074562	0.00	0.00	intersect
4	26.01	7.03	1.90	4000000	0.00	0.00	shade
5	3.72	7.30	0.27	100	2.71	73.03	calc_tile

The columns can be interpreted as follows:

- **%time**
- Percentage of the total execution time program spent in this function
- All functions combined should add up to 100%
- **cumulative seconds**
- Cumulative sum of exclusive runtimes of all functions up to and including this one.
- **self seconds**
- Number of seconds used by this function (exclusive).
- By default, the list is sorted according to this field.
- **calls** The number of times this function was called.
- **self ms/call** Average number of milliseconds per call that were spent in this function (exclusive).
- **total ms/call** Average number of milliseconds per call that were spent in this function, including its callees (inclusive).

Analysis - Example

- In the example above, optimization attempts would definitely start with the `intersect()` function, and `shade()` would also deserve a closer look.
- The corresponding exclusive percentages can hint at the maximum possible gain. If, e.g., `shade()` could be optimized to become twice as fast, the whole program would run in roughly 7.3 -0.95 = 6.35 seconds, i.e., about 15% faster.

Outcome of a profiling

- The outcome of a profiling run can depend crucially on the ability of the compiler to perform *function inlining.*
- *Inlining is an optimization technique that replaces* a function call by the body of the callee, reducing overhead.
- If inlining is allowed, the profiler output may be strongly distorted when some hot spot function gets inlined and its runtime is attributed to the caller.
- If the compiler/profiler combination has no support for correct profiling of inlined functions, it may be required to disallow inlining altogether.
- Of course, this may itself have some significant impact on program performance characteristics.

Flat Profile

- A flat profile contains a lot of information
- It does not reveal how the runtime contribution of a certain function is composed of several different callers
- Which other functions (callees) are called from it, and which contribution to runtime they in turn incur.
- This data is provided by the *butterfly graph, or callgraph profile:*

```
index % time    self  children    called     name
                0.27    7.03     100/100         main [2]
[1]     99.9    0.27    7.03     100         calc_tile [1]
                1.90    5.14 4000000/4000000     shade [3]
-----------------------------------------------
                                                 <spontaneous>
[2]     99.9    0.00    7.30                 main [2]
                0.27    7.03     100/100         calc_tile [1]
-----------------------------------------------
                             5517592             shade [3]
                1.90    5.14 4000000/4000000     calc_tile [1]
[3]     96.2    1.90    5.14 4000000+5517592 shade [3]
                5.14    0.00 26074562/26074562     intersect [4]
                             5517592             shade [3]
-----------------------------------------------
                5.14    0.00 26074562/26074562     shade [3]
[4]     70.2    5.14    0.00 26074562         intersect [4]
```

The columns can be interpreted as follows:

- **%time** The percentage of overall runtime spent in this function, including its callees (inclusive time).
- This should be identical to the product of the number of calls and the time per call on the flat profile.
- **self** For each indexed function, this is exclusive execution time (identical to flat profile).
- For its callers (callees), it denotes the inclusive time this function (each callee) contributed to each caller (this function).
- **children** For each indexed function, this is inclusive minus exclusive runtime, i.e., the contribution of all its callees to inclusive time.
- Part of this time contributes to inclusive runtime of each of the function's callers and is denoted in the respective caller rows.
- **called** denotes the number of times the function was called (probably split into recursive plus nonrecursive contributions).
- Which fraction of the number of calls came from each caller is shown in the caller row, whereas the fraction of calls for each callee that was initiated from this function can be found in the callee rows.

Example 2

```
#include<stdio.h>
void func4(void)
{
            printf("\n Inside func4() \n");
             for(int count=0;count<=0XFFFF;count++);
 }
void func3(void)
{
            printf("\n Inside func3() \n");
            for(int count=0;count<=0XFFFFFFF;count++);
}
void func2(void)
{
            printf("\n Inside func2() \n");
            for(int count=0;count<=0XFFF;count++); func3();
}
void func1(void)
{
            printf("\n Inside func1() \n");
            for(int count=0;count<=0XFFFFFF;count++); func2();
}
int main(void)
{
            printf("\n main() starts...\n");
            for(int count=0;count<=0XFFFFF;count++);

            func1();
            func4();
            printf("\n main() ends...\n");

            return 0;
}
```

Compile

- `gcc -pg -g program_name.c`
- `./a.out`
- `gprof a.out gmon.out`

Flat profile:

- Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
95.20	0.54	0.54	1	542.66	542.66	func3
5.29	0.57	0.03	1	30.15	572.81	func1
0.00	0.57	0.00	1	0.00	542.66	func2
0.00	0.57	0.00	1	0.00	0.00	func4

Call graph

```
granularity: each sample hit covers 2 byte(s) for 1.75% of 0.57 seconds

index % time    self  children    called     name
                0.03    0.54       1/1           main [2]
[1]    100.0    0.03    0.54       1          func1 [1]
                0.00    0.54       1/1           func2 [3]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00    0.57                  main [2]
                0.03    0.54       1/1           func1 [1]
                0.00    0.00       1/1           func4 [5]
-----------------------------------------------
                0.00    0.54       1/1           func1 [1]
[3]     94.7    0.00    0.54       1          func2 [3]
                0.54    0.00       1/1           func3 [4]
-----------------------------------------------
                0.54    0.00       1/1           func2 [3]
[4]     94.7    0.54    0.00       1          func3 [4]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[5]      0.0    0.00    0.00       1          func4 [5]
-----------------------------------------------
```

Line-based profiling

- Function profiling becomes useless when the program to be analyzed contains large functions (in terms of code lines)
- It contribute significantly to overall runtime:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
73.21	13.47	13.47	1	13.47	18.40	MAIN__
6.47	14.66	1.19	21993788	0.00	0.00	mvteil_
6.36	15.83	1.17	51827551	0.00	0.00	ran1_
6.25	16.98	1.15	35996244	0.00	0.00	gzahl_

- Here the MAIN function in a Fortran program requires over 73% of overall runtime but has about 1700 lines of code.
- If the hot spot in such functions cannot be found by simple common sense, tools for line-based profiling should be used.

Line-based profiling

- The number of sampling hits (first column) and the relative percentage of total program samples (second column):

```
1                    :        DO 215 M=1,3
2    4292   0.9317   :           bremsdir(M) = bremsdir(M) + FH(M)*Z12
3    1462   0.3174   : 215    CONTINUE
4                    :
5     682   0.1481   :        U12 = U12 + GCL12 * Upot
6                    :
7                    :        DO 230 M=1,3
8    3348   0.7268   :           F(M,I)=F(M,I)+FH(M)*Z12
9    1497   0.3250   :           Fion(M)=Fion(M)+FH(M)*Z12
10    501   0.1088   :230     CONTINUE
```

Line-based profiling

- The compiler-generated symbol tables must be consistent so that a machine instruction's address in memory can be properly matched to the correct source line.
- Modern compilers can reorganize code heavily if high optimization levels are enabled.
- Loops can be fused or split, lines rearranged, variables optimized away, etc., so that the actual code executed may be far from resembling the original source.
- Furthermore, due to the strongly pipelined architecture of modern microprocessors it is usually impossible to attribute a specific moment in time to a particular source line or even machine instruction.
- However, looking at line-based profiling data on a loop-by-loop basis (samples integrated across the loop body) is relatively safe;
- In case of doubt, recompilation with a lower optimization level (and inlining disabled) may provide more insight.

Hardware performance counters

- The first step in performance profiling is concerned with pinpointing the hot spots in terms of runtime, i.e., clock ticks.
- **Clock ticks are insufficient**
- But when it comes to identifying the actual reason for a code to be slow, or
- If one merely wants to know by which resource it is limited,.

- Modern processors feature a small number of *performance counters (<10), which are special on-chip* registers that get incremented each time a certain event occurs.

Number of bus transactions, i.e., cache line transfers.

- Events like “cache misses” are commonly used instead of bus transactions, however one should be aware that prefetching mechanisms (in hardware or software) can interfere with the number of cache misses counted.
- Bus transactions are often the safer way to account for the actual data volume transferred over the memory bus.
- If the memory bandwidth used by a processor is close to the theoretical maximum, there is no point in trying to optimize further for better bus utilization.
- The number can also be used for checking the correctness of some theoretical model one may have developed for estimating the data transfer requirements of an application

Number of loads and stores

- Together with bus transactions, this can give an indication as to how efficiently cache lines are used for computation.
- If, e.g., the number of DP loads and stores per cache line is less than its length in DP words, this may signal strided memory access.
- One should however take into account how the data is used;
- if, for some reason, the processor pipelines are stalled most of the time, or if many arithmetic operations on the data are performed out of registers, the strided access may not actually be a performance bottleneck.

Number of floating-point operations

- The importance of this very popular metric is often overrated.
- Data transfer is the dominant performance-limiting factor in scientific code.
- Even so, if the number of floating-point operations per clock cycle is somewhere near the theoretical maximum (either given by the CPU's peak performance, or less if there is an asymmetry between MULT and ADD operations), standard code optimization is unlikely to achieve any improvement and algorithmic changes are probably in order.

Mispredicted branches

- This counter is incremented when the CPU has predicted the outcome of a conditional branch and the prediction has proved to be wrong.
- Depending on the architecture, the penalty for a mispredicted branch can be tens of cycles.
- In general, scientific codes tend to be loop-based so that branches are well predictable.
- However, “pointer chasing” and computed branches increase the probability of mispredictions.

Pipeline stalls

- Dependencies between operations running in different stages of the processor pipeline can lead to cycles during which not all stages are busy, so-called *stalls or bubbles.*
- *Often bubbles cannot be* avoided, e.g., when performance is limited by memory bandwidth and the arithmetic units spend their time waiting for data.
- Hence, it is quite difficult to identify the point where there are “too many” bubbles.
- Stall cycle analysis is especially important on in-order architectures (like, e.g., Intel IA64) because bubbles cannot be filled by the hardware if there are no provisions for executing instructions that appear later in the instruction stream but have their operands available.

Number of instructions executed

- Together with clock cycles, this can be a guideline for judging how effectively the superscalar hardware with its multiple execution units is utilized.
- Experience shows that it is quite hard for compiler-generated code to reach more than 2–3 instructions per cycle, even in tight inner loops with good pipelining properties.

lipfpm profile

- In order to get a quick overview of the performance properties of an application, a simple tool can measure overall counts from start to finish and probably calculate some *derived metrics* like “instructions per cycle” or “cache misses per load or store.”
- A typical output of such a tool could look like this if run with some application code (these examples were compiled from output generated by the lipfpm tool,)

```
1  CPU Cycles.....                                  8721026107
2  Retired Instructions.....                        21036052778
3  Average number of retired instructions per cycle..... 2.398151
4  L2 Misses.....                                   101822
5  Bus Memory Transactions.....                     54413
6  Average MB/s requested by L2.....                2.241689
7  Average Bus Bandwidth (MB/s).....                1.197943
8  Retired Loads.....                               694058538
9  Retired Stores.....                              199529719
10 Retired FP Operations.....                       7134186664
11 Average MFLOP/s.....                             1225.702566
12 Full Pipe Bubbles in Main Pipe.....              3565110974
13 Percent stall/bubble cycles.....                 40.642963
```

lipfpm profile

- The large number of retired instructions per cycle indicates that the hardware is well utilized.
- The required bandwidths from the caches and main memory and the relation between retired load/store instructions to L2 cache misses.
- However, there are pipeline bubbles in 40% of all CPU cycles.
- It is hard to tell without some reference whether this is a large or a small value.

Example

- The bandwidth requirements, the low number of instructions per cycle, and the relation between loads/stores and cache misses indicate a memory-bound situation.
- In contrast to the previous case, the percentage of stalled cycles is more than doubled.
- Only an elaborate stall cycle analysis, based on more detailed metrics, would be able to reveal the origin of those bubbles.

```
1  CPU Cycles.....                                  28526301346
2  Retired Instructions.....                        15720706664
3  Average number of retired instructions per cycle.....  0.551095
4  L2 Misses.....                                   605101189
5  Bus Memory Transactions.....                     751366092
6  Average MB/s requested by L2.....                4058.535901
7  Average Bus Bandwidth (MB/s).....               5028.015243
8  Retired Loads.....                               3756854692
9  Retired Stores.....                              2472009027
10 Retired FP Operations.....                       4800014764
11 Average MFLOP/s.....                             252.399428
12 Full Pipe Bubbles in Main Pipe.....              25550004147
13 Percent stall/bubble cycles.....                 89.566481
```

Manual instrumentation

- If the overheads subjected to the application by standard compiler-based instrumentation are too large, or
- If only certain parts of the code should be profiled in order to get a less complex view on performance properties, manual instrumentation may be considered.
- The programmer inserts calls to a wallclock timing routine like `gettimeofday()`
- The results returned by timing routines should be interpreted Carefully.
- The most frequent mistake with code timings occurs when the time periods to be measured are in the same order of magnitude as the timer resolution, i.e., the minimum possible interval that can be resolved.

Common sense optimizations

- Very simple code changes can often lead to a significant performance boost.
- The most important “common sense” guidelines regarding the avoidance of performance pitfalls are summarized:
- **Do less work!**
- **Avoid Expensive Operations**
- **Shrink the workset**

Do less work!

- Rearranging the code such that less work than before is being done will improve performance.
- A very common example is a loop that checks a number of objects to have a certain property, but all that matters in the end is that *any object has the property at all:*

Do less work

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6   endif
7 enddo
```

- If complex_func() has no side effects, the only information that gets communicated to the outside of the loop is the value of FLAG.
- In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as FLAG changes

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6     exit
7   endif
8 enddo
```

Avoid expensive operations

- Sometimes, implementing an algorithm is done in a thoroughly "one-to-one" way, translating formulae to code without any reference to performance issues.
- While this is actually good (performance optimization always bears the slight danger of changing numerics, if not results), in a second step all those operations should be eliminated that can be substituted by "cheaper" alternatives.
- Prominent examples for such "strong" operations are trigonometric functions or exponentiation.
- Bear in mind that an expression like x **2.0 is often not optimized by the compiler to become x*x but left as it stands, resulting in the evaluation of an exponential and a logarithm.
- The corresponding optimization is called ***strength reduction***.
- *Apart from the* simple case described above, strong operations sometimes appear with a limited set of fixed arguments.

Avoid expensive operations

- This is an example from a simulation code for nonequilibrium spin systems:

```
1 integer :: iL,iR,iU,iO,iS,iN
2 double precision :: edelz,tt
3 ...                                  ! load spin orientations
4 edelz = iL+iR+iU+iO+iS+iN            ! loop kernel
5 BF    = 0.5d0*(1.d0+TANH(edelz/tt))
```

- The last two lines are executed in a loop that accounts for nearly the whole runtime of the application.
- The integer variables store spin orientations (up or down, i.e., -1 or +1, respectively), so the edelz variable only takes integer values in the range {-6, . . . , +6}.
- The tanh() function is one of those operations that take vast amounts of time (at least tens of cycles), even if implemented in hardware.

Avoid expensive operations – tanh()

- It is easy to eliminate the tanh() call completely by *tabulating* the function over the range of arguments required
- Assuming that `tt` does not change its value so that the table does only have to be set up once:

```
1 double precision, dimension(-6:6) :: tanh_table
2 integer :: iL,iR,iU,iO,iS,iN
3 double precision :: tt
4 ...
5 do i=-6,6                                  ! do this once
6   tanh_table(i) = 0.5d0*(1.d0+TANH(dble(i)/tt))
7 enddo
8 ...
9 BF = tanh_table(iL+iR+iU+iO+iS+iN) ! loop kernel
```

- The table look-up is performed at virtually no cost compared to the `tanh()` evaluation since the table will be available in L1 cache at access latencies of a few CPU cycles.
- Due to the small size of the table and its frequent use it will fit into L1 cache and stay there in the course of the calculation.

Shrink the working set

- The *working set of a code is the amount of memory it uses* in the course of a calculation, or at least during a significant part of overall runtime.
- Shrinking the working set by whatever means is a good thing because it raises the probability for cache hits.
- In the above example, the original code used standard four-byte integers to store the spin orientations.
- The working set was thus much larger than the L2 cache of any processor.
- By changing the array definitions to use `integer(kind=1)` for the spin variables, the working set could be reduced by nearly a factor of four, and became comparable to cache size.

Simple measures, large impact: Elimination of common subexpressions

- Common subexpression elimination is an optimization that is often considered a task for compilers.
- Basically one tries to save time by precalculating parts of complex expressions and assigning them to temporary variables before a code construct starts that uses those parts multiple times.
- In case of loops, this optimization is also called *loop invariant code motion:*

```
1  ! inefficient
2  do i=1,N
3    A(i)=A(i)+s+r*sin(x)
4  enddo
```



```
tmp=s+r*sin(x)
do i=1,N
  A(i)=A(i)+tmp
enddo
```

- A lot of compute time can be saved by this optimization, especially where “strong” operations (like sin()) are involved.
- Subexpressions are obstructed by other code and not easily recognizable, compilers are in principle able to detect this situation.
- Compute the subexpression out of the loop if this required employing associativity rules.
- A good strategy is to help the compiler by eliminating common subexpressions by hand.

Avoiding Branches

- "Tight" loops, i.e., loops that have few operations in them, are typical candidates for software pipelining, loop unrolling, and other optimization techniques.
- If for some reason compiler optimization fails or is inefficient, performance will suffer.
- This can easily happen if the loop body contains conditional branches:

```
1  do j=1,N
2    do i=1,N
3      if(i.ge.j) then
4        sign=1.d0
5      else if(i.lt.j) then
6        sign=-1.d0
7      else
8        sign=0.d0
9      endif
10     C(j) = C(j) + sign * A(i,j) * B(i)
11   enddo
12 enddo
```

Avoiding Branches

- In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored.
- The if statement serves to decide about which factor to use.
- Each time a corresponding conditional branch is encountered by the processor, some *branch prediction logic tries to guess the most probable* outcome of the test before the result is actually available, based on statistical methods.
- The instructions along the chosen path are then fetched, decoded, and generally fed into the pipeline.
- If the anticipation turns out to be false (this is called a *mispredicted branch or branch miss), the pipeline has to be flushed back to the position* of the branch, implying many lost cycles.

Avoiding Branches

- Fortunately, the loop nest can be transformed so that all if statements vanish:

```
1  do j=1,N
2    do i=j+1,N
3      C(j) = C(j) + A(i,j) * B(i)
4    enddo
5  enddo
6  do j=1,N
7    do i=1,j-1
8      C(j) = C(j) - A(i,j) * B(i)
9    enddo
10 enddo
```

Using SIMD instruction sets

- The use of SIMD in microprocessors is often termed “vectorization,”
- It is more similar to the multitrack property of modern vector systems.
- A “vectorizable” loop in this context will run faster if more operations can be performed with a single instruction
- The size of the data type should be as small as possible.
- Switching from DP to SP data could result in up to a twofold speedup, with the additional benefit that more items fit into the cache.
- Preferring SIMD instructions over scalar ones is no guarantee for a performance improvement.
- If the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.
- Register-to-register operations will be greatly accelerated, but this will only lengthen the time the registers wait for new data from the memory subsystem.

Using SIMD instruction sets

- A single precision ADD instruction was depicted that might be used in an array addition loop:

```
1 real, dimension(1:N) :: r, x, y
2 do i=1, N
3   r(i) = x(i) + y(i)
4 enddo
```

- All iterations in this loop are independent, there is no branch in the loop body, and the arrays are accessed with a stride of one.
- The use of SIMD requires some rearrangement of a loop kernel like the one above to be applicable:
- A number of iterations equal to the SIMD register size has to be executed as a single “chunk” without any branches in between.
- This is actually a well-known optimization that can pay off even without SIMD and is called *loop unrolling*.

Using SIMD instruction sets

- Since the overall number of iterations is generally not a multiple of the register size, some remainder loop is left to execute in scalar mode.

```
1  ! vectorized part
2  rest = mod(N,4)
3  do i=1,N-rest,4
4    load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
5    load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
6    ! "packed" addition (4 SP flops)
7    R3 = ADD(R1,R2)
8    store [r(i),r(i+1),r(i+2),r(i+3)] = R3
9  enddo
10 ! remainder loop
11 do i=N-rest+1,N
12   r(i) = x(i) + y(i)
13 enddo
```

- R1, R2, and R3 denote 128-bit SIMD registers here.
- In an optimal situation all this is carried out by the compiler automatically.
- Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.

SIMD Aligned and Unaligned

- SIMD instruction sets distinguish between *aligned and unaligned* data.
- If an aligned load or store is used on a memory address that is not a multiple of 16, an exception occurs.
- In cases where the compiler knows nothing about the alignment of arrays used in a vectorized loop
- Unaligned (or a sequence of scalar) loads and stores must be used, incurring some performance penalty.
- The programmer can force the compiler to assume optimal alignment, but this is dangerous if one cannot make absolutely sure that the assumption is justified.

Loop Dependency – SIMD Vectorization

- A loop with a true dependency cannot be SIMD vectorized.

```
1 do i=2,N
2   A(i)=s*A(i-1)
3 enddo
```

- The compiler will revert to scalar operations here, which means that only the lowest operand in the SIMD registers is used (on x86 architectures).

Loop Dependency – SIMD Vectorization

- One possible definition is that all arithmetic within the loop is executed using the full width of SIMD registers.
- The load and store instructions could still be scalar; compilers tend to report such loops as “vectorized” as well.
- On x86 processors with SSE support, the lower and higher 64 bits of a register can be moved independently.

Loop Dependency – SIMD Vectorization

- The vector addition loop above could thus look as follows in double precision:

```
1  rest = mod(N,2)
2  do i=1,N-rest,2
3    ! scalar loads
4    load R1.low = x(i)
5    load R1.high = x(i+1)
6    load R2.low = y(i)
7    load R2.high = y(i+1)
8    ! "packed" addition (2 DP flops)
9    R3 = ADD(R1,R2)
10   ! scalar stores
11   store r(i) = R3.low
12   store r(i+1) = R3.high
13 enddo
14 ! remainder "loop"
15 if(rest.eq.1) r(N) = x(N) + y(N)
```

Loop Dependency – SIMD Vectorization

- This version will not give the best performance if the operands reside in a cache.
- Although the actual arithmetic operations (line 9) are SIMD-parallel, all loads and stores are scalar.
- The only option to identify such a failure is manual inspection of the generated assembly code.
- If the compiler cannot be convinced to properly vectorize a loop even with additional command line options or source code directives, a typical “last resort” before using assembly language altogether is to employ *compiler intrinsics.*
- *Intrinsics are constructs that resemble assembly* instructions so closely that they can usually be translated 1:1 by the compiler.
- The user is relieved from the burden of keeping track of individual registers, because the compiler provides special data types that map to SIMD operands.
- Intrinsics are not only useful for vectorization but can be beneficial in all cases where high-level language constructs cannot be optimally mapped to some CPU functionality.
- Unfortunately, intrinsics are usually not compatible across compilers even on the same architecture.

The role of compilers: General optimization options

- Every compiler offers a collection of standard optimization options(-O0, -O1, ...).
- What kinds of optimizations are employed at which level is by no means standardized and often (but not always) documented in the manuals.
- All compilers refrain from most optimizations at level -O0, which is hence the correct choice for analyzing the code with a debugger.
- At higher levels, optimizing compilers mix up source lines, detect and eliminate “redundant” variables, rearrange arithmetic expressions, etc.,
- So that any debugger has a hard time giving the user a consistent view on code and data.

Inlining

- Inlining tries to save overhead by inserting the complete code of a function or subroutine at the place where it is called.
- Each function call uses up resources because arguments have to be passed, either in registers or via the stack (depending on the number of parameters and the calling conventions used).

Aliasing

- The compiler, guided by the rules of the programming language and its interpretation of the source, must make certain assumptions that may limit its ability to generate optimal machine code.
- The typical example:

```
1 void scale_shift(double *a, double *b, double s, int n) {
2   for(int i=1; i<n; ++i)
3     a[i] = s*b[i-1];
4 }
```

- Assuming that the memory regions pointed to by a and b do not overlap,
- The ranges [a,a+n-1] and [b,b+n-1] are disjoint, the loads and stores in the loop can be arranged in any order.
- The compiler can apply any software pipelining scheme it considers appropriate, or it could unroll the loop and group loads and stores in blocks, as shown in the following pseudocode:

```
1 loop:
2   load R1 = b(i+1)
3   load R2 = b(i+2)
4   R1 = MULT(s,R1)
5   R2 = MULT(s,R2)
6   store a(i) = R1
7   store a(i+1) = R2
8   i = i + 2
9   branch -> loop
```

Aliasing

- The C and C++ standards allow for arbitrary *aliasing of pointers.*
- *It* must thus be assumed that the memory regions pointed to by a and b do overlap.
- Lacking any further information, the compiler must generate machine instructions according to this scheme.
- Among other things, SIMD vectorization is ruled out.
- The processor hardware allows reordering of loads and stores within certain limits, but this can of course never alter the program's semantics.
- All C/C++ compilers have command line options to control the level of aliasing the compiler is allowed to assume (e.g., `-fno-fnalias` for the Intel compiler and `-fargument-noalias` for the GCC specify that no two pointer arguments for any function ever point to the same location).
- If the compiler is told that argument aliasing does not occur, it can in principle apply the same optimizations as in equivalent Fortran code.

Computational Accuracy

- Compilers sometimes refrain from rearranging arithmetic expressions if this required applying associativity rules, except with very aggressive optimizations turned on.
- The reason for this is the infamous nonassociativity of FP operations (a+b)+c is, in general, not identical to a+(b+c)
- if a, b, and c are finite-precision floating-point numbers.
- If accuracy is to be maintained compared to nonoptimized code, associativity rules must not be used and it is left to the programmer to decide whether it is safe to regroup expressions by hand.
- Modern compilers have command line options that limit rearrangement of arithmetic expressions even at high optimization levels.

Using compiler logs

- In order to make the decisions of the compiler's "intelligence" available to the user
- Many compilers offer options to generate annotated source code listings or at least logs that describe in some detail what optimizations were performed.
- Compiler log for a software pipelined triad loop is shown below.
- "Peak" indicates the maximum possible execution rate for the respective operation type on this architecture (MIPS R14000).

```
1   #<swps> 16383 estimated iterations before pipelining
2   #<swps>     4 unrollings before pipelining
3   #<swps>    20 cycles per 4 iterations
4   #<swps>     8 flops         ( 20% of peak) (madds count as 2)
5   #<swps>     4 flops         ( 10% of peak) (madds count as 1)
6   #<swps>     4 madds         ( 20% of peak)
7   #<swps>    16 mem refs      ( 80% of peak)
8   #<swps>     5 integer ops   ( 12% of peak)
9   #<swps>    25 instructions  ( 31% of peak)
10  #<swps>     2 short trip threshold
11  #<swps>    13 integer registers used.
12  #<swps>    17 float registers used.
```

Register optimizations

- It is one of the most vital, but also most complex tasks of the compiler to care about register usage.
- The compiler tries to put operands that are used “most often” into registers and keep them there as long as possible, given that it is safe to do so.
- If, e.g., a variable’s address is taken, its value might be manipulated elsewhere in the program via the address.
- In this case the compiler may decide to write a variable back to memory right after any change on it.
- Loop bodies with lots of variables and many arithmetic expressions are hard for the compiler to optimize because it is likely that there are too few registers to hold all operands at the same time.
- The number of integer and floating-point registers in any processor is strictly limited.
- Today, typical numbers range from 8 to 128.
- If there is a register shortage, variables have to be *spilled, i.e., written to memory, for* later use.
- If the code’s performance is determined by arithmetic operations, register spill can hamper performance quite a bit.
- In such cases it may even be worthwhile splitting a loop in two to reduce register pressure.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.