High Performance Computing - OpenMP

Lin Feng
Computer Science
University of Bristol
Candidate: 97302
rg18215@bristol.ac.uk

***Abstract*—This reports covers all the optimizations applied to make *d2q9-bgk.c* run faster. Tools such as OpenMP and techniques like vectorization are used to enhance the runtime and achieve better results.**

I. Optimization

A. Intel compiler options

Since there are many maths functions in collision step, flag *-fast* optimizes calculations by maximizing speed across the entire program. On Linux systems, it sets compiler options *-ipo, -O3, -no-prec-div, -static, -fp-model fast=2* and *-xHost*, flag *-Ofast* also implies options *-O3, -no-prec-div* and *-fp-model fast=2*. Considering *-Ofast* can be replaced by *-fast*.

When using the flag *-fast*, the -xHost option setting can be overridden by specifying a different processor-specific *-x* option on the command line and the last option specified on the command line takes precedence. For instance, if you specify option *-fast -xSSE3*, option *-xSSE3* takes effect. In contrast, if you specify *-xSSE3 -fast*, option *-xHost* takes effect [1]. *-xHost* tells the compiler to generate instructions for the highest instruction set available on the compilation host processor. As I know, BCp3 should support processor AVX and Intel MIC AVX-51. While only 1/16th of the SP float capability will be used as long as the program is not vectorized.

Furthermore, option *-static* prevents linking with shared libraries, it causes the executable to link all libraries statically [2]. However, when it is used with OpenMP, the warning message *"the use of 'mktemp' is dangerous, better use 'mkstemp'"* shows up.

Interprocedural optimization may be enabled by /Qipo (-ipo) across source files. This may give the compiler additional information about a loop, such as trip counts, alignment or data dependencies. It may also allow inlining of function calls [3].

Consequently, I chose to use flags *-iop, -Ofast* and *-xHost* while using OpenMP.

B. Avoid return value

The return value of a function will be stored in a register. If this return data has no intended usage, time and space are wasted in storing this information. The function should be defined as *void* to minimize the extra handling in the function.

I changed function ***accelerate_flow*** to a *void* function.

C. Loop unrolling

Unrolling the loops can be used to squeeze every last ounce of speed, but this may cost time of development or modification if the number of times the loop is executed has to change. Also, it is more difficult to do this when the loop is controlled by user input. Unrolling loops eliminate the increment for each loop iteration and avoids operation for the memory accesses on the variable index used by the array. In other hands, combining as many loops as possible also can decrease the unnecessary memory accesses and simplify the instruction.

I unrolled the loops for calculating the ***local_density*** and combined loops in ***propagate, rebound, collision*** and ***av_velocity*** steps into one loop.

D. Improving the arithmetic

In many cases, macros can be used to break up a piece of code into functions to eliminate typing duplicate code and to make things easily modifiable and readable. Since they are resolved at compile time, they require absolutely no runtime overhead. Also, replacing integer division with multiplies is another way to arrange and improve the arithmetic. For instance, change $x = a / b / c$ to $x = a / (b * c)$. It would probably not be a good trade-off if that line is executed only once, but if it is inside a loop that executes many times, it can really make a difference. Additionally, using a different expression that yields the same value but is cheaper to compute can affect the runtime. As an example, bit-and is cheaper than the remainder, like $x = y\%32$ can be represented as $x = y\&31$ [4]. Although many compilers will do this for you automatically.

I sort out the calculation in ***collision*** step and used macros to define the arithmetic.

II. Vectorization

The restrict qualifier will let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used provides the only means of accessing the memory in question in the scope in which the pointers live. If the loop vectorizes without using the restrict keyword, the compiler will be checking for aliasing at runtime. The compiler will not do any runtime checks for aliasing if you use the restrict keyword [3].

III. Parallelization - openMP

A. parallel for schedule

Schedule controls how loop iterations are divided among threads. Choosing the right schedule can have a great impact on the speed of the application.

static schedule means that iterations blocks are mapped statically to the execution threads in a loop. OpenMP runtime guarantees that two separate loops with the same number of iterations that run with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range(s) in both parallel regions. This is quite important on NUMA systems, if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop, the same thread could access the same memory location faster since it will reside on the same NUMA node. However, If each iteration takes vastly different from the meantime to be completed, then high work imbalance might occur in the static case. In this case, run with **dynamic** schedule might be more efficient but requires some communication overhead. Measure the runtime is the best way to choose which schedule clause to use in the program.

B. parallel for reduction

For loops which represent a reduction, for example, if parallelize the for loops by simply using **for**, it would cause a data race, because multiple threads could try to update the shared variable at the same time. Also if use **for** with **critical** to make sure update data correctly, threads can be idle while waiting to access a critical section. Since some threads may finish a piece of computation before others and have to wait for others to catch up. For avoiding waiting, OpenMP has the special **reduction** clause which can express the reduction of a **for** loop.

IV. Conclution

After optimization that I merged four loops in different functions into one loop, the runtime gets slightly faster, since the main loop cannot be vectorized still. By using OpenMP, I used **reduction** for summary calculation of ***av_vels*** and **schedule(static)** for the other loops. the runtime gets rapidly faster. The best runtime I've got is shown below.

	Optimization	OpenMP
128*128	19.1 s	1.9 s
256*256	155.0 s	12.6 s
1024*1024	644.0 s	56.5 +/- 2.0 s

References

[1] Software.intel.com. (2019). fast. [online] Available at: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-fast [Accessed 19 Feb. 2019].

[2] Software.intel.com. (2019). static. [online] Available at: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-static [Accessed 19 Feb. 2019].

[3] D3f8ykwhia686p.cloudfront.net. (2019). [online] Available at: https://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf [Accessed 19 Feb. 2019].

[4] Slideshare.net. (2019). Embedded C - Optimization techniques. [online] Available at: https://www.slideshare.net/EmertxeSlides/embedded-c-optimization-techniques [Accessed 19 Feb. 2019].