Optimize Data Structures and Memory Access Patterns to Improve Data Locality

<keywords: Intel Parallel Studio XE, VTune Amplifier XE, data locality>

Abstract

Cache is one of the most important resources of modern CPUs: it’s a smaller and faster part of the memory sub-system where copies of the most frequently used memory locations are stored. When data required by an instruction reside in the cache the instruction will executed immediately. Otherwise, execution of instructions might get suspended until the required data is fetched from the memory. Since copying data from the memory is a long latency operation, we’d like to minimize cache misses by designing algorithms and data structures to exploit data locality.

This article discusses symptoms of poor data locality, techniques to detect related performance bottlenecks, and possible optimizations to cure the problem.

This article is part of the larger series, "Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

A classical example that demonstrates the problem of poor data locality and its impact is a naïve implementation of matrix multiplication algorithm. Let’s start with serial version of the algorithm that multiplies square matrices A and B and stores result in matrix C according to the following formula: $C_{ij} = \sum A_{ik}B_{kj}$; each matrix is stored in an array in row-major order and the size of each matrix is quite big (NxN):

```c
void serial_mm( PType& A, PType& B, PType& C )
{
    for( size_t i = 0; i < N; ++i ) {
        for( size_t j = 0; j < N; ++j )
            for( size_t k = 0; k < N; k++ )
                C[i*N+j] += A[i*N+k] * B[j+k*N];
    }
}
```

This is a trivial example of the compute-intensive algorithm which shares common characteristics with many other algorithms: 1) it’s implemented using nested for-loops; and 2) it manipulates large chunk of data. Another, less obvious, characteristic is that the iterations of the outer loop are independent and it seems to be a perfect candidate for parallelization. The following code shows a parallel version of the algorithm implemented with Intel® Cilk™ Plus (we just replaced key-word for with Intel® Cilk Plus™ cilk_for and used the Intel® Compiler that supports Intel Cilk Plus language extension):

```c
void parallel_mm( PType& A, PType& B, PType& C, bool in_order )
{
    cilk_for( size_t i = 0; i < N; ++i ){
        for( size_t j = 0; j < N; ++j )
            for( size_t k = 0; k < N; k++ )
                C[i*N+j] += A[i*N+k] * B[j+k*N];
    }
}
```
Comparing performance of the parallel implementation vs. serial for big matrices, we discover that parallel version actually runs slower.

This is one of the examples where high-level analysis is not helpful in root causing the problem. Indeed, if we run Intel® VTune™ Amplifier XE Concurrency analysis we see that there is a problem: a poor concurrency level while running a hot-spot function (parallel_mm), but it doesn’t tell us why:

![Concurrency analysis screenshot](image)

We could get a similar result running Intel® Parallel Amplifier’s Concurrency analysis. However, using Intel VTune Amplifier XE, extreme edition of Intel Parallel Amplifier, adds one more dimension to the presentation of the performance statistics: the timeline view. The timeline view shows the application’s worker threads and their behavior over time. In case of the concurrency analysis applied to the matrix multiplication program, Intel® VTune™ Amplifier XE detected three worker threads created by Intel® Cilk™ Plus’ run-time system in addition to the main thread (marked as “Cilk Worker” on the timeline view), and distribution of the CPU Time per worker thread over time (shown in brown color on the timeline). Based on the data collected by Concurrency analysis we conclude that while CPU utilization by worker threads is very low, CPU Time is very high throughout the execution of the algorithm.

The next step in attempt to find the cause of the poor concurrency is to run one of the low-level profilers supported by Intel VTune Amplifier XE:
Intel VTune Amplifier XE supports several low-level analysis types that are based on Event-based Sampling (EBS): Lightweight Hotspots and a few analysis types grouped under the “Advanced” category. The purpose of EBS analysis is to collect hardware events by sampling special registers called PMUs (Performance Monitoring Units) which could be programmed to increment their values when a certain hardware event occurs (e.g. branch misprediction, cache miss etc). CPU can generate many different events and looking at one of them or a group might not always be helpful because often it is unclear whether the numbers that the profiler shows you are really the ones that are responsible for bad performance. Often, the right combination of events and a certain formula have to be used in order to confirm whether or not there is a particular performance issue in the application. If you run low-level profiler for the first time or you are not sure what type of problem you should be looking for, it is recommended that you start with “General Exploration.” “General Exploration” is a pre-configured analysis that collects multiple hardware events and uses a number of formulas which might help to understand what type of problem the application has: the tool collects hardware events, shows data that are the result of some formula, and highlights numbers that might indicate a performance problem. All events and formulas are described in the Intel VTune Amplifier XE Help in great detail; a shorter description can be seen on tooltips.
Running “General Exploration” for matrix multiplication program reveals several performance bottlenecks (cells marked with pink):

1. **CPI Rate**: Shows how many CPU cycles were spent per instruction on average. Ideally, we’d like this number to be less or equal to 1 but in our case it’s greater than 5.

2. **Retire Stalls**: Shows a ratio of the number of cycles when no micro-operations are retired to all cycles. In the absence of long latency operations and dependency chains, this number should be equal or greater than 1, in our case it’s ~0.8.

3. **LLC Miss**: Shows a ratio of cycles with outstanding LLC (Last-level Cache) misses to all cycles. Any memory requests that miss LLC must be serviced by local or remote DRAM, with significant latency. Ideally, this number should be very close to 0.

4. **Execution Stalls**: Shows a ratio of cycles with no micro-operations executed to all cycles. Execution Stalls are caused by waiting on critical computational resource.

A conclusion that can be made based on the data shown by Intel VTune Amplifier XE is that CPU spends time waiting for data to be fetched from a local or remote DRAM due to the high rate of LLC misses. When we double-click on the function, Intel VTune Amplifier XE will show a function source view and distribution of the hardware events per source line. For our example, most of the samples were generated by the inner-most for-loop:
Let's examine the data access pattern implemented in parallel_mm function for $i = 0$ and $j = 0$:

Each iteration of the inner-most loop touches an element of matrix A that is close to the element touched on the previous iteration and such an access pattern exploits efficient data locality. Access pattern for matrix B, however, is not efficient because on each iteration the program needs to access $j$-th element of $k$-th column where $k$ is an inner loop counter. Such an inefficient data access pattern is the reason of the program's poor performance because it leads to the high rate of cache misses.

Let's consider a more efficient implementation of the algorithm which requires only a small change in the code which dramatically improves the data access pattern:

```c
void parallel_mm( PType& A, PType& B, PType& C )
{
    // Note the changed loop order
    cilk_for( size_t i = 0; i < N; ++i ) {
```
for (size_t k = 0; k < N; ++k)
  for (size_t j = 0; j < N; j++)
    C[i*N+j] += A[i*N+k] * B[j+k*N];
}

Exchanging two inner loops leads to an improved data locality (i=0, k=0):

![Diagram showing improved data locality]

With this new implementation, for each i (counter of the outer loop) we calculate the entire row of matrix C and touch only consecutive elements of matrices A and B. This change greatly improves performance by increasing CPI and minimizing LLC misses:
Advice

Intel® VTune Amplifier XE has many useful features that can help find performance bottlenecks in your program. While high-level analysis types can often help with algorithmic optimizations, low-level analysis types focus on hardware behavior during the application execution. Modern CPUs have become very complicated and it might take an expert to perform low-level performance analysis. However, Intel VTune Amplifier XE’s philosophy is to simplify low-level performance analysis and make it more accessible to non-expert users. When high-level analysis types cannot help to figure out the problem, try running Intel VTune Amplifier XE General Exploration which might give you a hint as to what to do next and localize the problem.

In the previous section we discussed only one example of the data locality problem and the change of the algorithm helped us improve data locality and performance. However, poor data locality can be caused by poorly designed data structures and it might be impossible to improve data locality without changing a layout of data. This second type of problems related to a data locality will still have the same symptoms that can be detected by Intel VTune Amplifier XE: low concurrency levels, poor CPI, and a high rate of cache misses.

Design your data structures and algorithms with data locality in mind to achieve good performance and scalability. Both serial and parallel applications will benefit from exploiting a good data locality.

Usage Guidelines

Performance analysis is an iterative process. When using one of the Intel VTune Amplifier XE analysis types, re-run it after introducing changes to your code. Intel VTune Amplifier XE provides an automatic comparison feature so that you can compare two profiles for two different runs and track your progress that way.

In this article we discussed “General Exploration” analysis. Its purpose is to help you get started with the low-level performance analysis. There are other, more specific, analysis types available (“Memory Access,” “Cycles and uOps,” etc.) which can be run as a next step or when you know what type of performance bottleneck you are investigating, or if you need more specific details.

Additional Resources

Intel® Software Network Parallel Programming Community

Intel(R) VTune(TM) Amplifier XE product page