Project Completion Report

Performance Analysis and Optimization of a Multithreaded Application using Intel VTune

Debopama Basu

Roll Number: 152402006,
Indian Institute of Technology Palakkad,
Nila Campus, Kanjikode, Pudussery West,
Palakkad - 678623, Kerala, India

1 Introduction

In today's world of high-performance computing, applications are often designed to run multiple tasks at once using multithreading. This means a single program can create several threads that work in parallel—especially useful on modern multi-core processors. It's a powerful way to speed things up, particularly in areas like scientific simulations, data processing, and graphics rendering.

But writing efficient multithreaded programs isn't always easy. Developers face challenges like threads competing for the same resources, the overhead of keeping them in sync, and making sure memory is accessed efficiently.

That's where tools like Intel® VTune™ Profiler come in. VTune helps developers dig deep into how their applications perform. It shows detailed data on CPU usage, thread behavior, memory patterns, and how well the code makes use of the hardware. It supports different platforms and programming languages, making it a go-to tool for optimizing today's complex software.

In this project, we use VTune Profiler to analyze a multithreaded tiled matrix multiplication program. Tiling is a technique that boosts performance by working on smaller blocks of a matrix, which makes better use of the CPU cache. When combined with multithreading, this approach spreads the workload across multiple cores. Using VTune, we identify performance bottlenecks, examine how well the threads are working together, and explore ways to make the program even faster.

2 Overview of Intel VTune Analysis

The *Intel VTune Profiler* is an analysis and tuning tool that provides predefined analysis configurations to address performance-related questions.

This section provides a summary of several Intel VTune Profiler analysis types used to evaluate various performance aspects and uncover opportunities to better leverage available hardware resources.

VTune Profiler Analysis Types

The following list gives a brief description of each analysis type:

- **performance-snapshot**: Provides a quick overview of the application's performance and suggests next steps for deeper analysis.

- **hotspots**: Examines call paths to determine where the code spends most of its time, helping identify opportunities for algorithm tuning. This belongs to VTune Profiler's *Algorithm* analysis group.

- **threading**: Evaluates how effectively the application uses parallelism and CPU cores. It helps visualize thread parallelism, identify low concurrency, and detect serial bottlenecks. This is part of the *Parallelism* analysis group.

- **memory-consumption**: Analyzes memory usage by the application, including distinct memory objects and their allocation stacks. Categorized under the *Algorithm* analysis group.

- **uarch-exploration**: Investigates CPU microarchitectural bottlenecks affecting application performance. Included in the *Microarchitecture* analysis group.
- **memory-access**: Measures metrics related to memory access. Useful for memory-bound applications, it helps identify performance impacts across the memory hierarchy (e.g., cache, main memory), and potential NUMA-related issues. Part of the *Microarchitecture* analysis group.
- **hpc-performance**: Focuses on compute-intensive applications, analyzing CPU and GPU usage. Provides insights into OpenMP efficiency, memory access, and vectorization. Part of the *Parallelism* analysis group.
- **io**: Analyzes the usage of IO subsystems, CPUs, and processor buses. Belongs to the *Input and Output* analysis group.

3 Related Works

Performance optimization in multithreaded applications continues to be a crucial research area in modern computing. Numerous studies have explored how hardware-aware performance tuning and profiling tools can uncover bottlenecks and improve efficiency.

A notable contribution in this domain is Ami Murawka's work on performance analysis of parallel applications using Intel VTune. In his paper *Performance Analysis and Optimization of Parallel Applications* [1], Murawka emphasizes the importance of using profiling tools like Intel VTune to identify performance bottlenecks in parallel applications. His study focuses on the analysis of thread concurrency, memory usage, and hardware-level optimizations that significantly enhance the performance of parallel applications.

Other related works in this field include the pioneering work *A Top-Down Method for Performance Analysis and Counters Architecture* [2], which lays the groundwork for structured performance analysis using processor performance counters. This methodology greatly complements tools like Intel VTune Profiler.

Sato et al. in *On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models Using Intel VTune* [3] investigate the effectiveness of Intel VTune across various threading models. The study highlights how VTune can distinguish between programming paradigms in terms of performance impact.

In another study, *Enhancing Application Performance using Heterogeneous Memory Architectures on a Many-Core Platform* [4], the authors analyze how leveraging memory heterogeneity on many-core processors can significantly boost application performance. Their results also emphasize the importance of profiling tools in adapting to evolving hardware landscapes.

Additionally, *Performance Analysis of Intel Ivy Bridge and Intel Broadwell Microarchitectures Using Intel VTune Amplifier Software* [5] provides comparative insights into how VTune helps uncover microarchitectural differences across generations of Intel processors.

4 Methodology

The matrix multiplication implementation was optimized in multiple stages to exploit both parallelism and memory hierarchy more effectively. The goal was to accelerate a large-scale square matrix multiplication (3200×3200) by incrementally applying advanced techniques in concurrency and vectorization. The methodology followed these key steps:

1. **Baseline: Tiled Matrix Multiplication with Pthreads**
   We began with a multithreaded matrix multiplication using the `pthread` library. To improve cache performance, we applied standard tiling, dividing the computation into smaller blocks that fit within the L1, L2, and L3 caches. Each thread handled a range of rows, and 3-level nested tiles were used to reduce cache misses and enhance locality.

2. **Hierarchical Tiling Optimization**
   Building on the basic pthread implementation, we structured the tiles into a hierarchical pattern: L3 tiles (512), L2 tiles (128), and L1 tiles (64). This multilevel tiling improved data reuse and reduced bandwidth pressure between cache levels. This design mimics real-world hardware cache behavior and results in better performance, particularly for large matrices.

3. **Parallelization with OpenMP**
   The pthread-based model was then replaced with OpenMP for simpler thread management and portability. OpenMP directives allowed us to parallelize

outer tile loops with minimal code complexity. Dynamic scheduling was used to balance load across threads, especially for uneven workloads or system noise.

4. **OpenMP with SIMD Vectorization**
   Finally, to utilize the CPU's SIMD (Single Instruction, Multiple Data) units, we applied the `#pragma omp simd` directive to the innermost loop of the matrix multiplication. Additionally, the matrix was flattened into a 1D row-major format to enable contiguous memory access and improve vectorization efficiency. This combination of OpenMP for parallelism, SIMD for vector-level concurrency, and cache-aware tiling yielded significant performance gains.

Each step introduced a targeted optimization, leading to substantial improvements

5 Tiled Matrix Multiplication (Size 3200)

We implement a multithreaded tiled matrix multiplication algorithm for 3200×3200 matrices using C++ threads and a tile size of 128. Tiling improves cache locality, and multithreading distributes the computation workload across all available CPU cores.

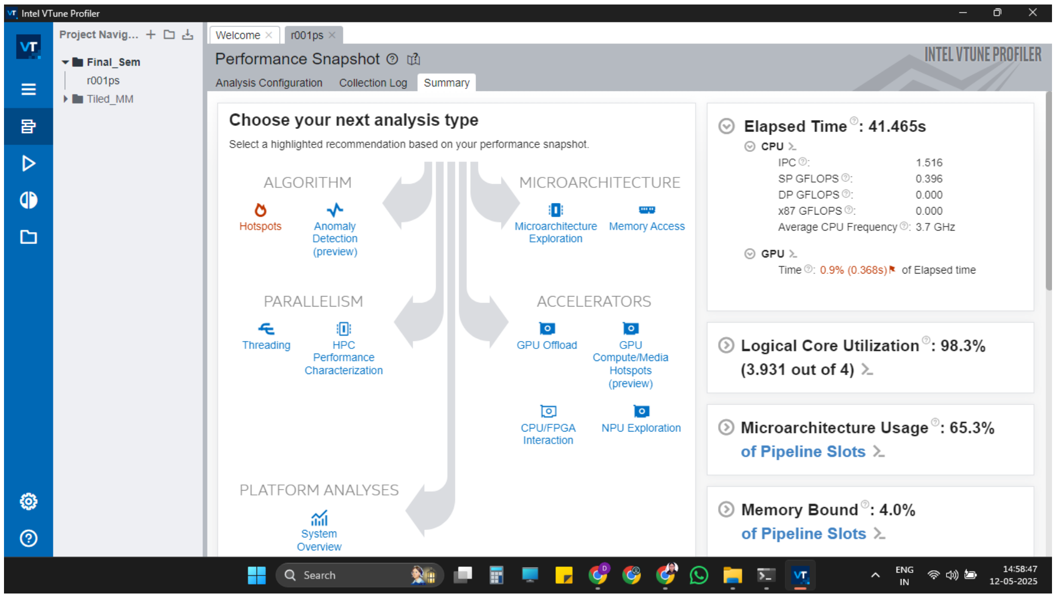
5.1 Implementation Highlights

- Matrices are stored as 2D vectors of type `float`.
- We use `std::thread` for parallelism, dividing matrix rows among available threads.
- Each thread computes a tiled block of rows of the output matrix.
- Mutex is used to synchronize logging to avoid output interleaving.
- Performance profiling is later done using Intel VTune to find hotspots and stalls.

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
#include <mutex>

std::mutex io_mutex;
const int N = 3200;
const int TILE_SIZE = 128;
const int NUM_THREADS = std::thread::hardware_concurrency();
using Matrix = std::vector<std::vector<float>>;

void initialize_matrix(Matrix &mat, float value) {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            mat[i][j] = value;
}

void tiled_multiply_worker(const Matrix &A, const Matrix &B, Matrix &C,
                           int row_start, int row_end) {
    for (int i = row_start; i < row_end; i += TILE_SIZE) {
        for (int j = 0; j < N; j += TILE_SIZE) {
            for (int k = 0; k < N; k += TILE_SIZE) {
                for (int ii = i; ii < std::min(i + TILE_SIZE, row_end); ++ii) {
                    for (int jj = j; jj < std::min(j + TILE_SIZE, N); ++jj) {
                        float sum = C[ii][jj];
                        for (int kk = k; kk < std::min(k + TILE_SIZE, N); ++kk) {
                            sum += A[ii][kk] * B[kk][jj];
                        }
                        C[ii][jj] = sum;
                    }
                }
            }
        }
    }
}

```

```cpp
void tiled_matrix_multiply(const Matrix &A, const Matrix &B, Matrix &C)
    {
    std::vector<std::thread> threads;
    int rows_per_thread = N / NUM_THREADS;

    for (int t = 0; t < NUM_THREADS; ++t) {
        int row_start = t * rows_per_thread;
        int row_end = (t == NUM_THREADS - 1) ? N : row_start +
            rows_per_thread;
        threads.emplace_back(tiled_multiply_worker, std::ref(A), std::
            ref(B),
                             std::ref(C), row_start, row_end);
    }

    for (auto &thread : threads)
        thread.join();
}

int main() {
    Matrix A(N, std::vector<float>(N));
    Matrix B(N, std::vector<float>(N));
    Matrix C(N, std::vector<float>(N, 0));

    initialize_matrix(A, 1.0f);
    initialize_matrix(B, 2.0f);

    auto start = std::chrono::high_resolution_clock::now();
    tiled_matrix_multiply(A, B, C);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Matrix␣multiplication␣completed␣in␣"
              << elapsed.count() << "␣seconds.\n";

    std::cout << "Detected␣hardware␣threads:␣"
              << NUM_THREADS << std::endl;

    return 0;
}
```

Listing 1: Tiled Matrix Multiplication with C++ Threads



Elapsed Time: 41.465s

CPU

```
IPC:                       1.516
SP GFLOPS:                 0.396
DP GFLOPS:                 0.000
x87 GFLOPS:                0.000
Average CPU Frequency:     3.7 GHz
```

GPU

Time: 0.9% (0.368s) of Elapsed time

Logical Core Utilization: 98.3% (3.931 out of 4)

Microarchitecture Usage: 65.3% of Pipeline Slots

Memory Bound: 4.0% of Pipeline Slots

Figure 1: Performance Snapshot showing different Analysis types in Intel VTune

5.2 Code Explanation

The program uses the following strategy:

1. **Matrix Initialization:** Both input matrices `A` and `B` are initialized with constant values for reproducibility.

2. **Tiling:** Matrix multiplication is performed using tiles to improve cache locality and reduce memory bandwidth pressure.

3. **Multithreading:** Work is split across threads, where each thread handles a specific range of matrix rows. This enhances parallelism and CPU utilization.

4. **Computation Loop:** The core of the matrix multiplication is in the line:

   ```
   sum += A[ii][kk] * B[kk][jj];
   ```

   This instruction is identified via VTune as a performance hotspot.

5. **VTune Profiling:** The application is analyzed using VTune for instruction pipeline behavior, stalls, and vectorization bottlenecks.

6 Hierarchical 3-Level Tiled Matrix Multiplication

To better utilize the CPU cache hierarchy (L1, L2, L3), we introduce a 3-level tiling strategy for matrix multiplication. This implementation improves data locality, reduces cache misses, and increases throughput by assigning distinct row blocks to multiple threads using C++'s standard threading.

6.1 Optimized Multithreaded 3-Level Tiled Matrix Multiplication

```cpp
...

const int N = 3200;          // Matrix size
const int TILE_L1 = 64;      // Fits in L1 cache of size 160 KB
const int TILE_L2 = 128;     // Fits in L2 cache of size 2.5 MB
const int TILE_L3 = 512;     // Fits in L3 cache of size 6.0 MB
const int NUM_THREADS = std::thread::hardware_concurrency();

...

// 3-level tiled matrix multiplication per thread
void tiled_multiply_worker(const Matrix &A, const Matrix &B, Matrix &C,
    int row_start, int row_end) {
    {
        std::lock_guard<std::mutex> lock(io_mutex);
        std::cout << "Thread " << std::this_thread::get_id()
                  << " started: rows " << row_start << " to " << row_end
                     - 1 << std::endl;
    }

    for (int i3 = row_start; i3 < row_end; i3 += TILE_L3) {
        for (int j3 = 0; j3 < N; j3 += TILE_L3) {
            for (int k3 = 0; k3 < N; k3 += TILE_L3) {

                for (int i2 = i3; i2 < std::min(i3 + TILE_L3, row_end);
                    i2 += TILE_L2) {
                    for (int j2 = j3; j2 < std::min(j3 + TILE_L3, N); j2
                        += TILE_L2) {
                        for (int k2 = k3; k2 < std::min(k3 + TILE_L3, N)
                            ; k2 += TILE_L2) {

```

```
                            for (int i1 = i2; i1 < std::min(i2 + TILE_L2, row_end); i1 += TILE_L1) {
                                for (int j1 = j2; j1 < std::min(j2 + TILE_L2, N); j1 += TILE_L1) {
                                    for (int k1 = k2; k1 < std::min(k2 + TILE_L2, N); k1 += TILE_L1) {

                                        for (int i = i1; i < std::min(i1 + TILE_L1, row_end); ++i) {
                                            for (int j = j1; j < std::min(j1 + TILE_L1, N); ++j) {
                                                float sum = C[i][j];
                                                for (int k = k1; k < std::min(k1 + TILE_L1, N); ++k) {
                                                    sum += A[i][k] * B[k][j];
                                                }
                                                C[i][j] = sum;
                                            }
                                        }
...
    {
        std::lock_guard<std::mutex> lock(io_mutex);
        std::cout << "Thread " << std::this_thread::get_id() << " finished.\n";
    }
}

// Spawns threads
void tiled_matrix_multiply(const Matrix &A, const Matrix &B, Matrix &C) {
    ...
}

int main() {
    ...
}
```

Listing 2: Tiled Matrix Multiplication with C++ Threads

6.2 Explanation and Optimization Benefits

This implementation significantly improves performance due to the following enhancements:

- **Hierarchical Tiling:** The code divides the computation into three levels of tiles: TILE_L1 (64), TILE_L2 (128), and TILE_L3 (512), which align well with L1, L2, and L3 cache sizes, respectively based on the cache size specifications of the used system.
- **Cache Efficiency:** Tiling increases spatial and temporal locality, reducing cache misses and memory latency.
- **Multithreading:** Rows are divided among hardware threads to fully utilize CPU cores, improving parallel throughput.
- **Reduced Memory Bandwidth Pressure:** Accesses to submatrices are more predictable and cache-friendly, lowering traffic to main memory.
- **Scalability:** The approach scales well with the number of cores due to even row distribution and minimized synchronization.

Overall, this approach provides substantial performance gains over naïve or single-level tiled methods, particularly on systems with deep cache hierarchies.

7 OpenMP-Accelerated 3-Level Tiled Matrix Multiplication

This version of tiled matrix multiplication uses OpenMP to parallelize the computation over multiple CPU threads while maintaining a 3-level cache-aware tiling scheme. This enhances both data locality and parallel throughput with minimal programmer effort.

7.1 OpenMP-Based Cache-Efficient Matrix Multiplication

```
...
void tiled_matrix_multiply(const Matrix &A, const Matrix &B, Matrix &C)
    {
    #pragma omp parallel for collapse(2) schedule(dynamic)
    for (int i3 = 0; i3 < N; i3 += TILE_L3) {
        ...
                                                C[i][j] = sum;
                                            }
...
}
```

Listing 3: Tiled Matrix Multiplication with C++ Threads

7.2 Performance Benefits and Explanation

This implementation is designed for high performance on multicore systems and leverages both thread-level and data-level parallelism:

- **OpenMP Parallelism:** Uses `#pragma omp parallel for` to parallelize the outermost loop, distributing large chunks of matrix computation across all available cores. Based on the system specification, collapse(2) is used as the system is having 4 threads running parallely.
- **3-Level Tiling:** Operates with cache-aware blocking to reduce L1, L2, and L3 cache misses. The innermost loops work on small matrix tiles that fit in the respective cache levels.
- **Dynamic Scheduling:** `schedule(dynamic)` allows better load balancing when iterations vary in execution time, which can happen due to memory access patterns.
- **Vectorization Potential:** Each innermost loop (involving element-wise operations) benefits from SIMD-level hardware vectorization due to predictable, aligned memory access.
- **Simplified Thread Management:** OpenMP abstracts thread creation, load distribution, and synchronization, leading to simpler and cleaner code compared to manual `std::thread`.

This version is highly portable, scalable, and efficient on modern multicore CPUs. It combines software-level control (tiling) with OpenMP's hardware-level threading efficiency.

8 SIMD-Aware Multilevel Tiled OpenMP Matrix Multiplication

This implementation performs high-performance matrix multiplication of two large 3200×3200 matrices using the following optimizations:

- **Flattened matrix representation** for better cache locality.
- **Three-level cache-aware tiling** to exploit L1, L2, and L3 caches efficiently.
- **OpenMP parallelism** using `#pragma omp parallel for collapse(2)`.
- **SIMD vectorization** on the innermost loop using `#pragma omp simd`.

8.1 SIMD based tiled matrix multiplication

```
...

using Matrix = std::vector<float>;  // Flattened matrix (row-major)

...

void tiled_matrix_multiply(const Matrix &A, const Matrix &B, Matrix &C)
    {
    #pragma omp parallel for collapse(2) schedule(dynamic)
    for (int i3 = 0; i3 < N; i3 += TILE_L3) {
        ...
        sum = C[idx(i, j)];
                                      #pragma omp simd
                                                for (int k = k1; k < std
                                                    ::min(k1 + TILE_L1, N
                                                    ); ++k) {
                                                    sum += A[idx(i, k)]
                                                        * B[idx(k, j)];
                                                }

                                                C[idx(i, j)] = sum;
                                            }
                                        ...
}

int main() {
    ...
}
```

Listing 4: Tiled Matrix Multiplication with C++ Threads

8.2 Performance Benefits and Explanation

- **Flattened Matrix:** By storing matrices in a single-dimensional array in row-major order, we reduce pointer indirection overhead and improve memory access patterns for CPU caching. Less cache thrashing due to better spatial locality.
- **Three-Level Tiling:** The matrix is divided into L3-, L2-, and L1-sized tiles that match typical cache sizes, minimizing cache misses.
- **OpenMP Parallelism:** The outermost loops are parallelized using OpenMP, which distributes tile computations across available threads.
- **SIMD Vectorization:** The innermost loop is vectorized using `#pragma omp simd`, allowing the CPU to process multiple elements in a single instruction parallely, improving arithmetic throughput. Higher Instruction level parallelism is observed

9 Experimental Results

Metric	Tiled MM (Pthread)	Tiled MM (Hierarchical + Pthread)	Tiled MM (OpenMP)	Tiled MM (OpenMP + SIMD)
Elapsed Time (s)	150.765	141.671	26.132	248.554
IPC	1.559	1.609	1.305	0.994
Logical Core Utilization	91.6% (3.663/4)	95.4% (3.816/4)	98.9%(3.956/4)	102%(4.089/4)
Microarchitecture Usage	66.9%	69.0%	51.5%	42.1%
Memory Bound	3.8%	3.4%	8.5%	16.5%

Table 1: Performance Snapshot analysis of Tiled Matrix Multiplication Implementations

Type	CPI	Retiring	Bad Speculation	Frontend Bound	Backend Bound	Memory Bound	Core Bound	Time (s)
MM	0.70	69.5%	3.4%	15.5%	11.6%	3.7%	7.9%	29.551
MM + OpenMP	0.59	56.6%	3.4%	17.8%	22.2%	13.3%	9.3%	10.369
MM + SIMD	0.89	64.2%	2.6%	17.1%	16.1%	6.7%	9.4%	14.861
MM + OpenMP + SIMD	1.02	46.2%	3.1%	14.5%	36.1%	24.5%	11.6%	**9.215**

Table 2: Microarchitectural Analysis of matrix multiplication implementations

10 Conclusion

Among the four evaluated matrix multiplication optimization strategies, the **MM + OpenMP + SIMD** version demonstrates the most effective performance in terms of execution time. Despite a higher *Cycles Per Instruction (CPI)* and lower *Retiring* percentage (46.2%), this configuration achieves the shortest elapsed time of just **9.215 seconds**, which is a substantial improvement over the other methods.

This result suggests that vectorization through SIMD, combined with OpenMP-based parallelism, significantly accelerates computation. The observed increase in back-end bound and core utilization indicates that the computation units are effectively saturated, albeit at the cost of reduced instruction efficiency. Nevertheless, for high-performance scenarios where execution speed is critical, this trade-off proves worthwhile.

Thus, **MM + OpenMP + SIMD** is the most suitable optimization approach when minimizing CPU time is the primary objective.

References

[1] A. Murawka, "Performance Analysis and Optimization of Parallel Applications," *Journal of High Performance Computing*, vol. 40, no. 5, pp. 1342–1360, 2017, `https://doi.org/10.1007/s10766-017-0495-0`.

[2] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, 2014, doi: `10.1109/ISPASS.2014.6844459`.

[3] M. Sato *et al.*, "On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models Using Intel VTune," *International Journal of Parallel Programming*, vol. 40, no. 5, pp. 539–565, 2012, doi: `10.1007/s10766-011-0182-6`.

[4] A. Jha *et al.*, "Enhancing Application Performance using Heterogeneous Memory Architectures on a Many-Core Platform," *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 38–45, 2016, doi: `10.1109/ICPPW.2016.29`.

[5] A. Alaghi, H. R. Zarandi, and M. Nazari, "Performance Analysis of Intel Ivy Bridge and Intel Broadwell Microarchitectures Using Intel VTune Amplifier Software," *2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)*, pp. 73–77, 2018, doi: `10.1109/CFIS.2018.8399107`.