ADVANCE COMPUTER ARCHITECTURE   
fINAL REPORT

COMPUTATION INTENSIVE CODE PERFORMANCE COMPARISON ON SIMD AND MIMD BASED ARCHITECTURES

Submitted to: Respectful Dr. Syed Abbas

Submitted by:

|  |  |  |  |
| --- | --- | --- | --- |
| Adeel Ahmad | Rana Aurangzaib | Zahid Hussain | Zubair Idrees |
| 197302 | 197304 | 197306 | 197301 |

Contents

[**Problem Statement** 2](#_Toc43329074)

[**Approach** 2](#_Toc43329075)

[**TEST BED MACHINES** 2](#_Toc43329076)

[MACHINE A (MIMD) 3](#_Toc43329077)

[MACHINE B (SIMD) 3](#_Toc43329078)

[**Code snippet & Brief Explanation** 4](#_Toc43329079)

[**Analysis** 5](#_Toc43329080)

[CPU Utilization 5](#_Toc43329081)

[Code Hot Spot And Related Assembly 6](#_Toc43329082)

[CPU Microarchitecture Level Analysis 7](#_Toc43329083)

[Memory Bound And Vectorization Analysis 9](#_Toc43329084)

[GPU Utilization 10](#_Toc43329085)

[**Inferences and conclusions** 11](#_Toc43329086)

[**Appendix** 12](#_Toc43329087)

## **Problem Statement**

Data scientist or Machine learning enthusiast has been trying to elicit performance of their learning models at scale will at some point hit a cap and start to experience various degrees of processing lag. Tasks that take minutes with smaller training sets may now take more hours—in some cases weeks—when datasets get larger. You need the best hardware.

In this report we will be evaluating the performance of simple but computationally intensive application on SIMD (Single Instruction Multiple Data Stream) and MIMD (Multiple Instruction Multiple Data Stream) based architecture to find out the best hardware for the application. Moreover, we will analyze our application is CPU or GPU bound and how effectively it offloads code to the GPU.

## **Approach**

We are going to create a simple but computation intensive program, execute it on both SIMD and MIMD architecture-based machines and further evaluate the performance objectively. There are many tools available in the market in order to evaluate performance of application depending on the test machine computer architecture. We have selected “Intel® VTune™ Profiler” as our test machine CPUs are of Intel.

Intel® VTune™ Profiler will help us in determining:

* The most time-consuming (hot) functions in application and/or on the whole system
* Sections of code that do not effectively utilize available processor time
* The best sections of code to optimize for sequential performance and for threaded performance
* Synchronization objects that affect the application performance
* Whether, where, and why your application spends time on input/output operations
* Whether your application is CPU or GPU bound and how effectively it offloads code to the GPU
* The performance impact of different synchronization methods, different numbers of threads, or different algorithms
* Thread activity and transitions

## **TEST BED MACHINES**

Followings are the details of the machines which we will be using to analyze the performance of the program. The specifications of both the machines are kept same, except the only difference is the GPU which is part of Machine B. Therefore, in next sections of report, “MACHINE A (MIMD)” refers to “MIMD” based architecture and “MACHINE B” refers to “SIMD” architecture.

### MACHINE A (MIMD)

#### Platform Info

Operating System: Microsoft Windows 10

RAM: 12GB DDR4

L1 Data Cache: 4 x 32 KBytes, 8-way set associative, 64-byte line size

L1 Instruction Cache: 4 x 32 KBytes, 8-way set associative, 64-byte line size

L2 Cache: 4 x 256 KBytes, 4-way set associative, 64-byte line size

L3 Cache: 6 MBytes, 12-way set associative, 64-byte line size

#### CPU

Name: Intel(R) Processor code named Kabylake

Frequency: 2.8 GHz

Logical CPU Count: 8

### MACHINE B (SIMD)

#### Platform Info

Operating System: Microsoft Windows 10

RAM: 12GB DDR4

L1 Data Cache: 4 x 32 KBytes, 8-way set associative, 64-byte line size

L1 Instruction Cache: 4 x 32 KBytes, 8-way set associative, 64-byte line size

L2 Cache: 4 x 256 KBytes, 4-way set associative, 64-byte line size

L3 Cache: 6 MBytes, 12-way set associative, 64-byte line size

#### CPU

Name: Intel(R) Processor code named Kabylake

Frequency: 2.8 GHz

Logical CPU Count: 8

#### GPU

Name: NVIDIA GeForce GTX 1050

Vendor: NVIDIA

Driver: 26.21.14.4122 (07-Nov-19)

Stepping: 0

EU Count: 24

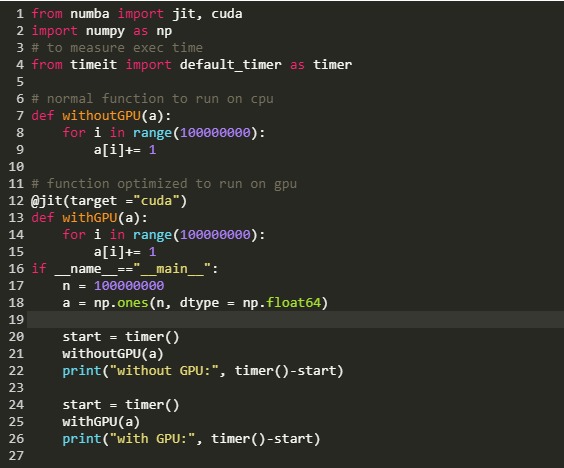
Max EU Thread Count: 7

Max Core Frequency: 1.1 GHz

## **Code snippet & Brief Explanation**

Here we have created two methods, in both of them, we are initializing a floating-point array of size **100 million** with ***ones***, further we are doing addition operation on every index of the array. Please note that as our focus is to evaluate SIMD and MIMD architecture quantitatively hence we have kept our application code quite simple but computation intensive.

Code Snippet:



We are using ***numba.jit*** decorator to offload computation over the GPU. The decorator has several parameters but we will work with only the *target* parameter. Target tells the ***jit*** to compile program code for either (“CPU” or “Cuda”). “Cuda” corresponds to GPU. However, if CPU is passed as an argument then the ***jit*** tries to optimize the code to run faster on CPU and improves the speed too.

However, it must be noted that the array is first copied from RAM to the GPU for processing and if the function returns anything then the returned values will be copied from GPU to CPU back. Therefore, for small data sets the speed of CPU is comparatively faster as there is no offloading from CPU to GPU.

## **Analysis**

Now we will execute our program and analyze the performance of the code using Intel VTune Profiler, on both the machines i.e. Machine A (MIMD) and Machine B (SIMD). We will look into below five main areas for performance analysis:

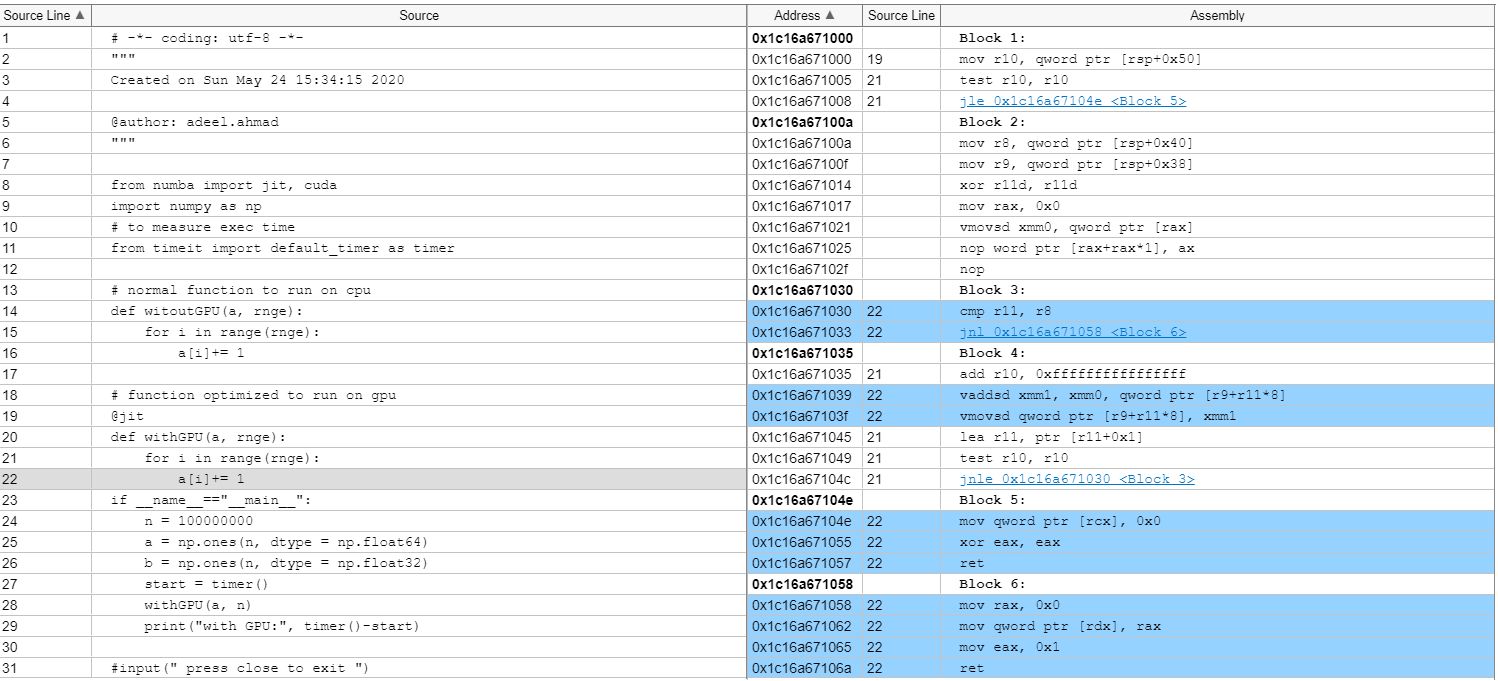
* 1. CPU Utilization
  2. Code Hotspot and related Assembly
  3. CPU Micro Architecture Level analysis
  4. Memory Bound and Vectorization Analysis
  5. GPU Utilization

### CPU Utilization

|  |  |
| --- | --- |
| **MACHINE A (MIMD)** | **MACHINE B (SIMD)** |
|  |  |
| As we can see from the comparison above **Effective CPU Time** is reduced **25 times** by executing code on GPU based architecture, CPI rate is increased which is expected as CPU waits for the results after passing instruction/data to the GPU. **So, SpeedUp is 25 times.** | |
|  |  |
| This histogram shows a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time add to the idle CPU utilization value. We can observe overall Elapsed time of program on CPU-1 in Machine A is 60s and Machine B is around 2.5s whereas CPU-0 remains same. | |

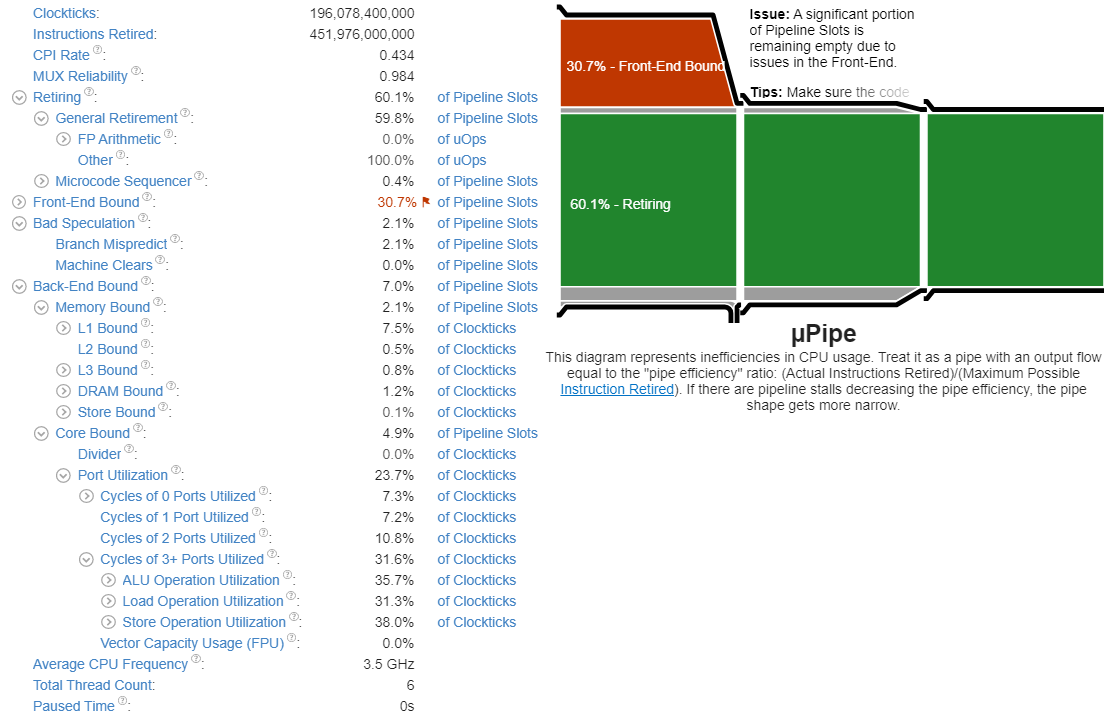
### Code Hot Spot And Related Assembly

The most time-consuming python code and corresponding assembly code is highlighted in the snapshot below. We can see utilization of Vector based instructions in GPU.

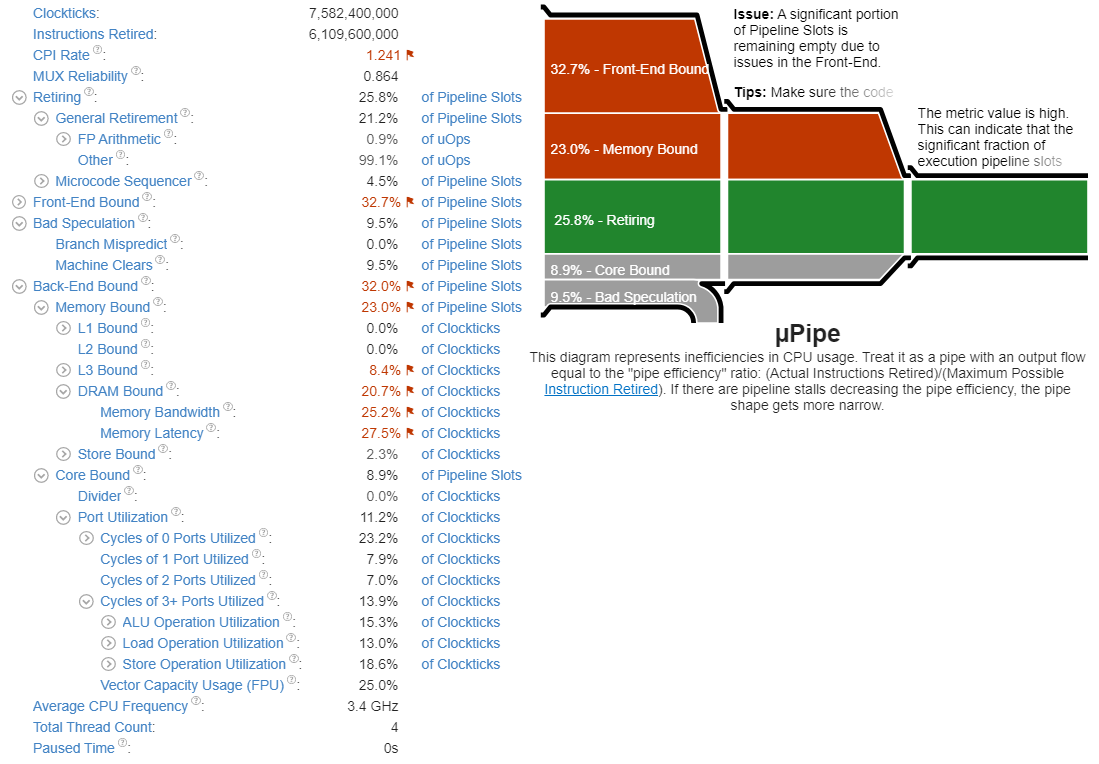


### CPU Microarchitecture Level Analysis

#### Machine A (MIMD)

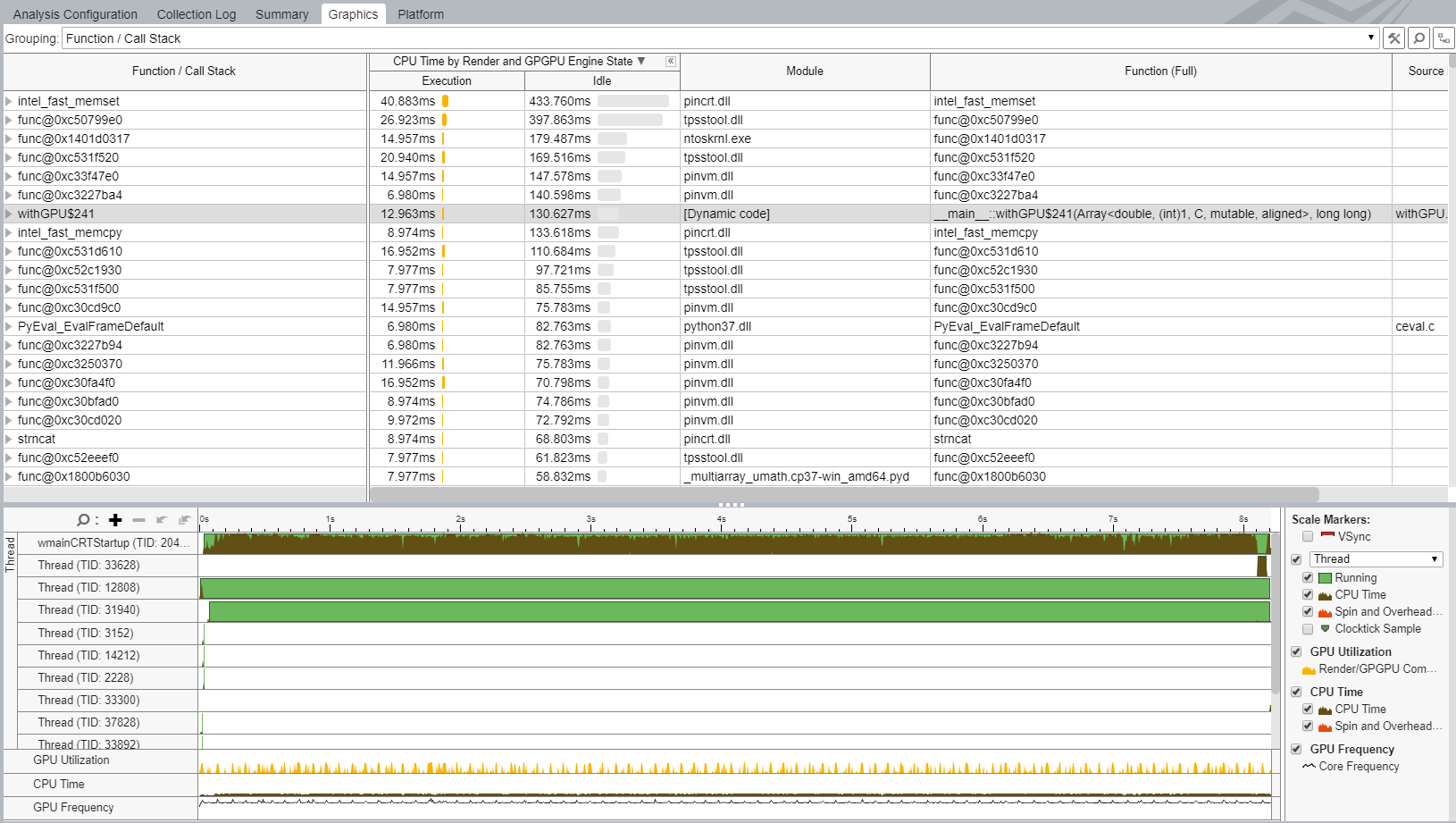


#### Machine B (SIMD)



### Memory Bound And Vectorization Analysis

|  |  |
| --- | --- |
| **MACHINE A (MIMD)** | **MACHINE B (SIMD)** |
|  |  |
| We observe that **“Memory Bound”** in Machine B is higher than Machine A. This indicates that the significant fraction of execution pipeline slots stalled due to demand of memory load and stores. It is because of offloading compute intensive instructions from CPU to GPU. GPU Utilization | |



## **Inferences and conclusions**

* It is evident from the snapshots above that the **Clockticks are 196 billion on “MACHINE A (MIMD)” and only 7.5 billion on “MACHINE B (SIMD)”** which is our GPU based machine. Even the CPI rate is high in GPU based machine, but **Clockticks** and **Instruction Retired** (number of instructions executed) are reduced that’s why we are getting better performance.
* CPI can be misleading, so we should understand the pitfalls. CPI (latency) is not the only factor affecting the performance of our code on system. The other major factor is the number of instructions executed (sometimes called path length). All optimizations or changes we make to our code will affect either the time to execute instructions (CPI) or the number of instructions to execute, or both. Using CPI without considering the number of instructions executed can lead to an incorrect interpretation of the results. In MACHINE B (SIMD), as we instructed our program to exploit GPU based architecture so the compiler vectorized our code and converted our math operations to operate on multiple pieces of data at once (SIMD). This would have the effect of replacing many single-data math instructions with fewer multiple-data math instructions. This reduced the number of instructions executed overall in our code on MACHINE B (SIMD), but it raised our CPI because multiple-data instructions are more complex and take longer to execute. This vectorization increased our performance, even though CPI went up.
* We observe that **“Memory Bound”** in Machine 2 is high. This indicates that the significant fraction of execution pipeline slots stalled due to demand of memory load and stores.
* Superscalar processors can be conceptually divided into the 'front-end', where instructions are pre-fetched and decoded into the operations that constitute them; and the 'back-end', where the required computation is performed. We observe that “Front-End Bound” is almost same in both machines that is around 30% whereas “Back-End Bound” higher in Machine B (SIMD) due to offloading the work to GPU. Nonetheless, we are still getting higher performance on Machine B.
* The **GPU Utilization** section above shows the time (in seconds) used by GPU. Note that GPU may work in parallel and the total time taken by GPU engines does not necessarily equal to the application Elapsed time. We can correlate that GPU utilization time with the overall Elapsed Time metric. The GPU Time value shows a share of the Elapsed time used by a particular GPU engine. We observe that overall Elapsed time is 2.11s on Machine B in which GPU took around 0.872s in parallel execution to CPU. So the GPU is taking around 40% from the overall Elapsed time.
* On Machine B took, total 10% of the GPU was utilized and our method took only 12.963 milliseconds to execute one hundred million instructions.

A note of caution: It is important to be aware of your total instructions executed as well. If suppose instructions retired is remaining fairly constant, CPI can be a good indicator of performance. If both the number of instructions and CPI are changing, we should look at both metrics to understand why performance increased or decreased.

##### additional information on Frontend Bound & Backend Bound

Superscalar processors can be conceptually divided into the 'front-end', where instructions are fetched and decoded into the operations that constitute them; and the 'back-end', where the required computation is performed. Each cycle, the front-end generates up to four of these operations placed into pipeline slots that then move through the back-end. Thus, for a given execution duration in clock cycles, it is easy to determine the maximum number of pipeline slots containing useful work that can be retired in that duration. The actual number of retired pipeline slots containing useful work, though, rarely equals this maximum. This can be due to several factors: some pipeline slots cannot be filled with useful work, either because the front-end could not fetch or decode instructions in time ('Front-end bound' execution) or because the back-end was not prepared to accept more operations of a certain kind ('Back-end bound' execution). Moreover, even pipeline slots that do contain useful work may not retire due to bad speculation. Front-end bound execution may be due to a large code working set, poor code layout, or microcode assists. Back-end bound execution may be due to long-latency operations or other contention for execution resources. Bad speculation is most frequently due to branch misprediction.

A significant proportion of pipeline slots are remaining empty. When operations take too long in the back-end, they introduce bubbles in the pipeline that ultimately cause fewer pipeline slots containing useful work to be retired per cycle than the machine is capable of supporting. This opportunity cost results in slower execution. Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port (for example, more multiply operations arriving in the back-end per cycle than the execution unit can support)

## **Appendix**

|  |  |
| --- | --- |
| CPU Time | CPU Time is time during which the CPU is actively executing your application. |
| Effective Time | Effective Time is CPU time spent in the user code. This metric does not include Spin and Overhead time |
| Microarchitecture Usage | Microarchitecture Usage metric is a key indicator that helps estimate (in %) how effectively your code runs on the current microarchitecture. Microarchitecture Usage can be impacted by long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end. |
| Retiring Metric | Retiring metric represents a Pipeline Slots fraction utilized by useful work, meaning the issued uOps that eventually get retired. Ideally, all Pipeline Slots would be attributed to the Retiring category. Retiring of 100% would indicate the maximum possible number of uOps retired per cycle has been achieved. Maximizing Retiring typically increases the Instruction-Per-Cycle metric. Note that a high Retiring value does not necessary mean no more room for performance improvement. For example, Microcode assists are categorized under Retiring. They hurt performance and can often be avoided. |
| Cycles per Instruction Retired, or CPI, | Cycles per Instruction Retired, or CPI, is a fundamental performance metric indicating approximately how much time each executed instruction took, in units of cycles. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best CPI of 0.25. But various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed CPI up. A CPI < 1 is typical for instruction bound code, while a CPI > 1 may show up for a stall cycle bound application, also likely memory bound. CPI is an excellent metric for judging an overall potential for application performance tuning. |
| General Retirement | This metric represents a fraction of slots during which CPU was retiring uOps not originated from the Microcode Sequencer. This correlates with the total number of instructions executed by the program. A uOps-per-Instruction ratio of 1 is expected. While this is the most desirable of the top 4 categories, high values may still indicate areas for improvement. If possible focus on techniques that reduce instruction count or result in more efficient instructions generation such as vectorization. |
| FP Arithmetic | This metric represents an overall arithmetic floating-point (FP) uOps fraction the CPU has executed (retired). |
| Micro Sequencer | This metric represents a fraction of slots during which CPU was retiring uOps fetched by the Microcode Sequencer (MS) ROM. The MS is used for CISC instructions not fully decoded by the default decoders (like repeat move strings), or by microcode assists used to address some modes of operation (like in Floating-Point assists). |
| Front-End Bound | Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End. Front-End denotes the first part of the processor core responsible for fetching operations that are executed later on by the Back-End part. Within the Front-End, a branch predictor predicts the next address to fetch, cache-lines are fetched from the memory subsystem, parsed into instructions, and lastly decoded into micro-ops (uOps). Front-End Bound metric denotes unutilized issue-slots when there is no Back-End stall (bubbles where Front-End delivered no uOps while Back-End could have accepted them). For example, stalls due to instruction-cache misses would be categorized as Front-End Bound. |
| Bad Speculation | Bad Speculation represents a Pipeline Slots fraction wasted due to incorrect speculations. This includes slots used to issue uOps that do not eventually get retired and slots for which the issue-pipeline was blocked due to recovery from an earlier incorrect speculation. For example, wasted work due to mispredicted branches is categorized as a Bad Speculation category. Incorrect data speculation followed by Memory Ordering Nukes is another example. |
| Branch Mispredicts | When a branch mispredicts, some instructions from the mispredicted path still move through the pipeline. All work performed on these instructions is wasted since they would not have been executed had the branch been correctly predicted. This metric represents slots fraction the CPU has wasted due to Branch Misprediction. These slots are either wasted by uOps fetched from an incorrectly speculated program path, or stalls when the out-of-order part of the machine needs to recover its state from a speculative path. |
| Machine Clears | Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measures three such events: memory ordering violations, self-modifying code, and certain loads to illegal address ranges. Machine Clears metric represents slots fraction the CPU has wasted due to Machine Clears. These slots are either wasted by uOps fetched prior to the clear, or stalls the out-of-order portion of the machine needs to recover its state after the clear. |
| Back-End Bound | Back-End Bound metric represents a Pipeline Slots fraction where no uOps are being delivered due to a lack of required resources for accepting new uOps in the Back-End. Back-End is the portion of the processor core where an out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to the program order. For example, stalls due to data-cache misses or stalls due to the divider unit being overloaded are both categorized as Back-End Bound. Back-End Bound is further divided into two main categories: Memory Bound and Core Bound |
| Memory Bound | This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline. |
| L1 Bound | This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. |
| L2 Bound | This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance. |
| L3 Bound | This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance. |
| DRAM Bound | This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance. |
| Memory Bandwidth | This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems. |
| Memory Latency | This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler). |
| Store Bound | This metric shows how often CPU was stalled on store operations. Even though memory store accesses do not typically stall out-of-order CPUs; there are few cases where stores can lead to actual stalls. |
| Core Bound | This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations). |
| Cache Bound | This metric shows how often the machine was stalled on L1, L2, and L3 caches. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data. |
| Vectorization | This metric represents the percentage of packed (vectorized) floating point operations. 0% means that the code is fully scalar. The metric does not take into account the actual vector length that was used by the code for vector instructions. So if the code is fully vectorized and uses a legacy instruction set that loaded only half a vector length, the Vectorization metric shows 100%. |
| SP FLOPS | The metric represents the percentage of single precision floating point operations from all operations executed by the applications. Use the metric for rough estimation of a SP FLOP fraction. If FMA vector instructions are used the metric may overcount. |
| DP FLOPS | The metric represents the percentage of double precision floating point operations from all operations executed by the applications. Use the metric for rough estimation of a DP FLOP fraction. If FMA vector instructions are used the metric may overcount. |
| X87FLOPs | The metric represents the percentage of x87 floating point operations from all operations executed by the applications. Use the metric for rough estimation of an x87 fraction. If FMA vector instructions are used the metric may overcount. |
| Non-FP | This metric represents the percentage of non-floating point operations executed by the application. Use the metric for rough estimation of a non-FP fraction. If FMA vector instructions are used the metric may overcount. |
| FP Arith/Mem Wr Instr. Ratio | This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 indicates unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution. |

|  |  |
| --- | --- |
| Clockticks per Instructions Retired | Clockticks per Instructions Retired (CPI) event ratio, also known as Cycles per Instructions, is one of the basic performance metrics for the hardware event-based sampling collection, also known as Performance Monitoring Counter (PMC) analysis in the sampling mode. This ratio is calculated by dividing the number of unhalted processor cycles (Clockticks) by the number of instructions retired. On each processor the exact events used to count clockticks and instructions retired may be different, but VTune Profiler knows the correct ones to use. |
| Retiring Metric | Retiring metric represents a Pipeline Slots fraction utilized by useful work, meaning the issued uOps that eventually get retired. Ideally, all Pipeline Slots would be attributed to the Retiring category. Retiring of 100% would indicate the maximum possible number of uOps retired per cycle has been achieved. Maximizing Retiring typically increases the Instruction-Per-Cycle metric. Note that a high Retiring value does not necessarily mean no more room for performance improvement. For example, Microcode assists are categorized under Retiring. They hurt performance and can often be avoided. |
| Cycles per Instruction Retired, or CPI, | Cycles per Instruction Retired, or CPI, is a fundamental performance metric indicating approximately how much time each executed instruction took, in units of cycles. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best CPI of 0.25. But various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed CPI up. A CPI < 1 is typical for instruction bound code, while a CPI > 1 may show up for a stall cycle bound application, also likely memory bound. CPI is an excellent metric for judging an overall potential for application performance tuning. |
| General Retirement | This metric represents a fraction of slots during which CPU was retiring uOps not originated from the Microcode Sequencer. This correlates with the total number of instructions executed by the program. A uOps-per-Instruction ratio of 1 is expected. While this is the most desirable of the top 4 categories, high values may still indicate areas for improvement. If possible, focus on techniques that reduce instruction count or result in more efficient instructions generation such as vectorization. |
| FP Arithmetic | This metric represents an overall arithmetic floating-point (FP) uOps fraction the CPU has executed (retired). |
| Micro Sequencer | This metric represents a fraction of slots during which CPU was retiring uOps fetched by the Microcode Sequencer (MS) ROM. The MS is used for CISC instructions not fully decoded by the default decoders (like repeat move strings), or by microcode assists used to address some modes of operation (like in Floating-Point assists). |
| Front-End Bound | Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End. Front-End denotes the first part of the processor core responsible for fetching operations that are executed later on by the Back-End part. Within the Front-End, a branch predictor predicts the next address to fetch, cache-lines are fetched from the memory subsystem, parsed into instructions, and lastly decoded into micro-ops (uOps). Front-End Bound metric denotes unutilized issue-slots when there is no Back-End stall (bubbles where Front-End delivered no uOps while Back-End could have accepted them). For example, stalls due to instruction-cache misses would be categorized as Front-End Bound. |
| Bad Speculation | Bad Speculation represents a Pipeline Slots fraction wasted due to incorrect speculations. This includes slots used to issue uOps that do not eventually get retired and slots for which the issue-pipeline was blocked due to recovery from an earlier incorrect speculation. For example, wasted work due to mispredicted branches is categorized as a Bad Speculation category. Incorrect data speculation followed by Memory Ordering Nukes is another example. |
| Branch Mispredicts | When a branch mispredicts, some instructions from the mispredicted path still move through the pipeline. All work performed on these instructions is wasted since they would not have been executed had the branch been correctly predicted. This metric represents slots fraction the CPU has wasted due to Branch Misprediction. These slots are either wasted by uOps fetched from an incorrectly speculated program path, or stalls when the out-of-order part of the machine needs to recover its state from a speculative path. |
| Machine Clears | Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measure three such events: memory ordering violations, self-modifying code, and certain loads to illegal address ranges. Machine Clears metric represents slots fraction the CPU has wasted due to Machine Clears. These slots are either wasted by uOps fetched prior to the clear, or stalls the out-of-order portion of the machine needs to recover its state after the clear. |
| Back-End Bound | Back-End Bound metric represents a Pipeline Slots fraction where no uOps are being delivered due to a lack of required resources for accepting new uOps in the Back-End. Back-End is the portion of the processor core where an out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to the program order. For example, stalls due to data-cache misses or stalls due to the divider unit being overloaded are both categorized as Back-End Bound. Back-End Bound is further divided into two main categories: Memory Bound and Core Bound |
| Memory Bound | This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline. |
| L1 Bound | This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. |
| L2 Bound | This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance. |
| L3 Bound | This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance. |
| DRAM Bound | This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance. |
| Memory Bandwidth | This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems. |
| Memory Latency | This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler). |
| Store Bound | This metric shows how often CPU was stalled on store operations. Even though memory store accesses do not typically stall out-of-order CPUs; there are few cases where stores can lead to actual stalls. |
| Core Bound | This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations). |
| Core Bound | This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations). |
| Cache Bound | This metric shows how often the machine was stalled on L1, L2, and L3 caches. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data. |
| Vectorization | This metric represents the percentage of packed (vectorized) floating point operations. 0% means that the code is fully scalar. The metric does not take into account the actual vector length that was used by the code for vector instructions. So if the code is fully vectorized and uses a legacy instruction set that loaded only half a vector length, the Vectorization metric shows 100%. |
| SP FLOPS | The metric represents the percentage of single precision floating point operations from all operations executed by the applications. Use the metric for rough estimation of a SP FLOP fraction. If FMA vector instructions are used the metric may overcount. |
| DP FLOPS | The metric represents the percentage of double precision floating point operations from all operations executed by the applications. Use the metric for rough estimation of a DP FLOP fraction. If FMA vector instructions are used the metric may overcount. |
| X87FLOPs | The metric represents the percentage of x87 floating point operations from all operations executed by the applications. Use the metric for rough estimation of an x87 fraction. If FMA vector instructions are used the metric may overcount. |
| Non-FP | This metric represents the percentage of non-floating point operations executed by the application. Use the metric for rough estimation of a non-FP fraction. If FMA vector instructions are used the metric may overcount. |
| FP Arith/Mem Wr Instr. Ratio | This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 indicates unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution. |