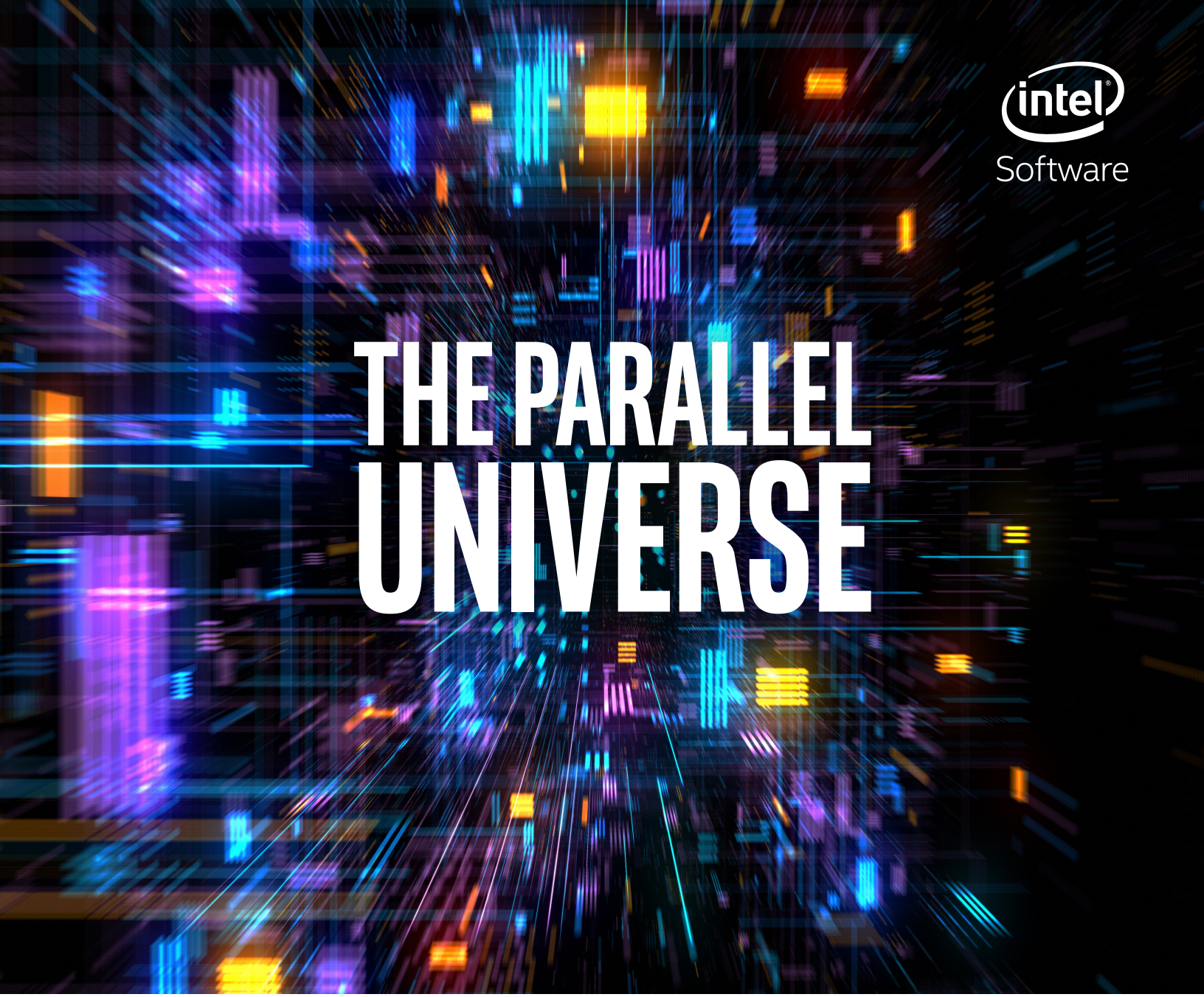


Accelerating XGBoost* for Intel® Xeon® Processors

Detecting and Mitigating False Sharing in Multi-Processors

Speeding Up Simulation Analysis with yt* and Intel® Distribution for Python*

Issue
38
2019

CONTENTS

Sign up for future issues

LETTER FROM THE EDITOR

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, is a longtime high-performance and parallel computing practitioner who has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

See You at the Intel® HPC Developer Conference

Come meet us in Denver and tell us what topics you want us to explore next year...and beyond

Before we get started, I have a couple of news items to pass along. First, the **Intel® HPC Developer Conference** returns to Denver, Colorado, November 17 and 18, just before SC19. Second, there's a new book you'll want to check out: **Pro TBB: C++ Parallel Programming with Threading Building Blocks.** It was written by Intel's Michael Voss, Rafael Asenjo from the University of Malaga, and the editor emeritus of *The Parallel Universe,* James Reinders. If you'll be in Denver next month, stop by the developer conference or the Intel booth at SC19. I'd like to hear your thoughts on the magazine and topics you want us to cover next year. You can also meet the authors of the book and learn more about TBB.

In this issue, our feature article, **Accelerated XGBoost* for Intel® Xeon® Processors**, describes a series of optimizations that dramatically improve the performance of the popular XGBoost machine learning library.

Next, we have **Detecting and Mitigating False Sharing in Multi-Processors**. False sharing is a subtle performance bug that limits parallel scalability. This article shows you how to detect and fix this problem using **Intel® VTune™ Amplifier**.

If you straddle the worlds of HPC and data analytics, you'll want to read **Speeding Up Simulation Analysis with yt* and Intel® Distribution for Python**, written with our collaborators at the Leibniz Supercomputing Centre. This article shows how they improved post-processing performance using the **Intel® Distribution for Python** and yt, "a community-developed analysis and visualization toolkit for volumetric data."

If you're interested in improving the security of your software, we have an article on **Intel® Software Guard Extensions** that describes the secure enclave model and provides simple code examples illustrating how enclaves are created and used.

We close out this issue with two editorials: one from Dennis O'Connell, Senior Director of Performance Engineering at Verizon and another from me.

The first one discusses how Verizon maximizes customer satisfaction using programming tools like **Intel® Parallel Studio XE** and **Intel® System Studio**.

You may remember this article from over two years ago: **Julia: A High-Level Language for Supercomputing** (*The Parallel Universe*, **Issue 29**). We close this issue with an editorial I wrote about **Composable Threading Coming to Julia***. Composability allows a program to spawn threads freely without worrying about oversubscribing the hardware because the runtime scheduler sorts everything out. It's a powerful new feature of the language. It's hard to predict whether Julia will succeed in the saturated marketplace of programming languages, but with over 10 million downloads and 3,000 packages, it appears to be gaining popularity. I'll try to get an updated article from the Julia developers for a future issue of *The Parallel Universe*.

As always, don't forget to check out **Tech.Decoded** for more information on Intel's solutions for code modernization, visual computing, data center and cloud computing, data science, and systems and IoT development.

Hard to believe, but this is our last issue of 2019. We're looking forward to bringing you lots more in 2020.

Henry A. Gabb
October 2019

ACCELERATING XGBOOST* FOR INTEL® XEON® PROCESSORS

How to Maximize Processor Performance for Machine Learning

Egor Smirnov, Software Engineering Manager, Intel Corporation

Gradient boosting[1] has many real-world applications as a general-purpose, supervised learning technique for regression, classification, and page ranking problems. The algorithm earned its fame in Kaggle* platform machine learning competitions, where it was recognized as the most popular machine learning algorithm. It's a common choice for large problems with a gradient-boosting model, with a histogram tree-building method[2] that helps to reduce training time without losing accuracy.

Training implementation of this method is quite complex because:

- **It contains many kernels** that impact execution time.
- **It doesn't use BLAS*/LAPACK*** or other common functions that are already highly optimized for many architectures.
- **There are many things that require specific optimization techniques** like irregular memory accesses, loops with dependencies, branch miss-prediction, etc.

This was a reason why optimizations for the XGBoost* library were limited before version 1.0. Intel has since made many optimizations to maximize performance during XGBoost training.

Measuring Performance Gains

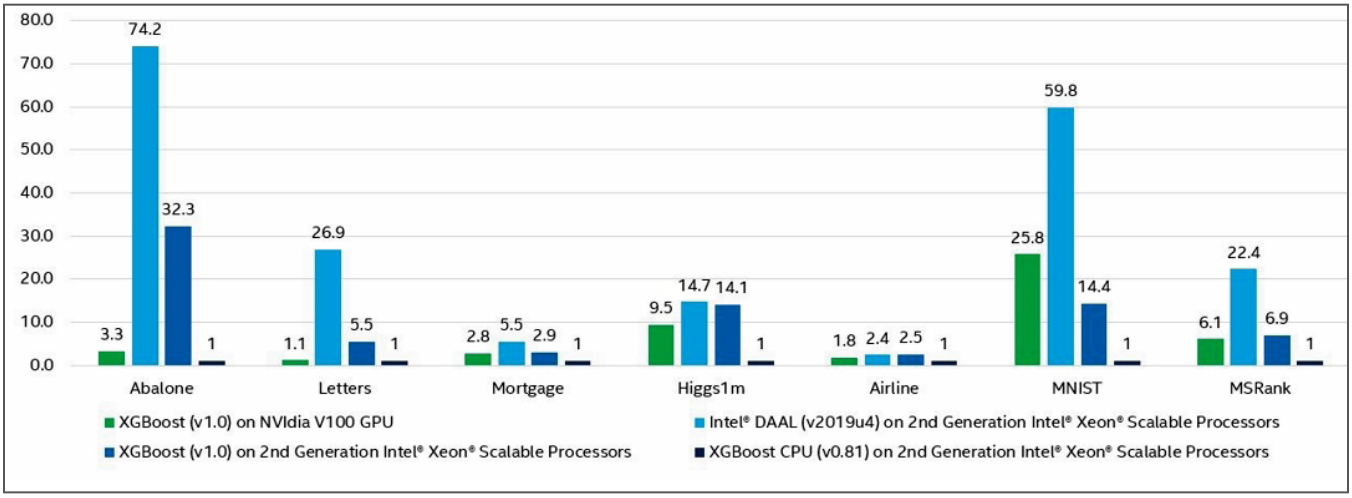
Table 1 compares XGBoost 0.81 to XGBoost 1.0. We added measurements with **Intel® Data Analytics Acceleration Library (Intel® DAAL)**,[3] a highly optimized library for Intel® CPUs, to show the potential gains that can be available in future XGBoost versions. **Figure 1** shows the gradient boosting training speedup versus the XGBoost baseline.

Table 1. XGBoost 0.81 and XGBoost 1.0 comparison

Version/ Data Set	Abalone	Letters	Mortgage	Higgs1m	Airline	MNIST	MSRank	Synthetic
XGB CPU (0.81), s	29.1	71.5	71.5	299.5	173	2,000	892	264.2
XGB CPU (1.0), s	1.3	18.9	18.6	19.8	75	90	127	12.8
Intel® DAAL, s	0.3	2.6	9.6	13.5	73	19	33	7.7
XGB GPU (1.0), s	7.4	52.7	20.7	23.7	79	41.9	122.2	16.1

We can see from **Table 1** that the:

- **Performance gain for the new XGBoost version is 11x on average**, depending on the number of cores, training parameters, and data set dimensions.
- **XGBoost performance is quite close to a state-of-the-art library** in terms of performance on Intel® Xeon® processors with Intel DAAL.
- **Intel Xeon processor implementation** is competitive with the GPU implementation.



1 **Gradient boosting training: Speedup versus the XGBoost CPU baseline, v0.81 (higher is better). For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.**

What Makes XGBoost Faster on Intel® Xeon® Processors?

The original source of the optimizations is Intel DAAL. Most of its techniques were moved to XGBoost. Let's take a deep dive into these changes to understand the source of the performance improvements. First, we'll prepare a breakdown for all functions in XGBoost 0.81 version (**Table 2**). Next, we'll look at the same functions, focusing on the percentage of training time (**Table 3**). We can see that most of the time is taken by the `BuildHist` function, so let's take a closer look at histogram building.

Table 2. Time spent in XGBoost 0.81 functions (See Appendix for system configuration details.)

Data Set/ Function	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, s	3	219	9	30	1	3
MSRank, s	8	673	11	81	6	51
Mortgage, s	12	101	3	117	31	11

Table 3. Percentage of training time in XGBoost 0.81 functions

Data Set/ Function	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, %	1.0	82.2	3.4	11.3	0.2	1.1
MSRank, %	0.9	80.3	1.3	9.6	0.7	6.1
Mortgage, %	4.4	35.8	1.2	41.2	10.8	4.0

Building Histograms

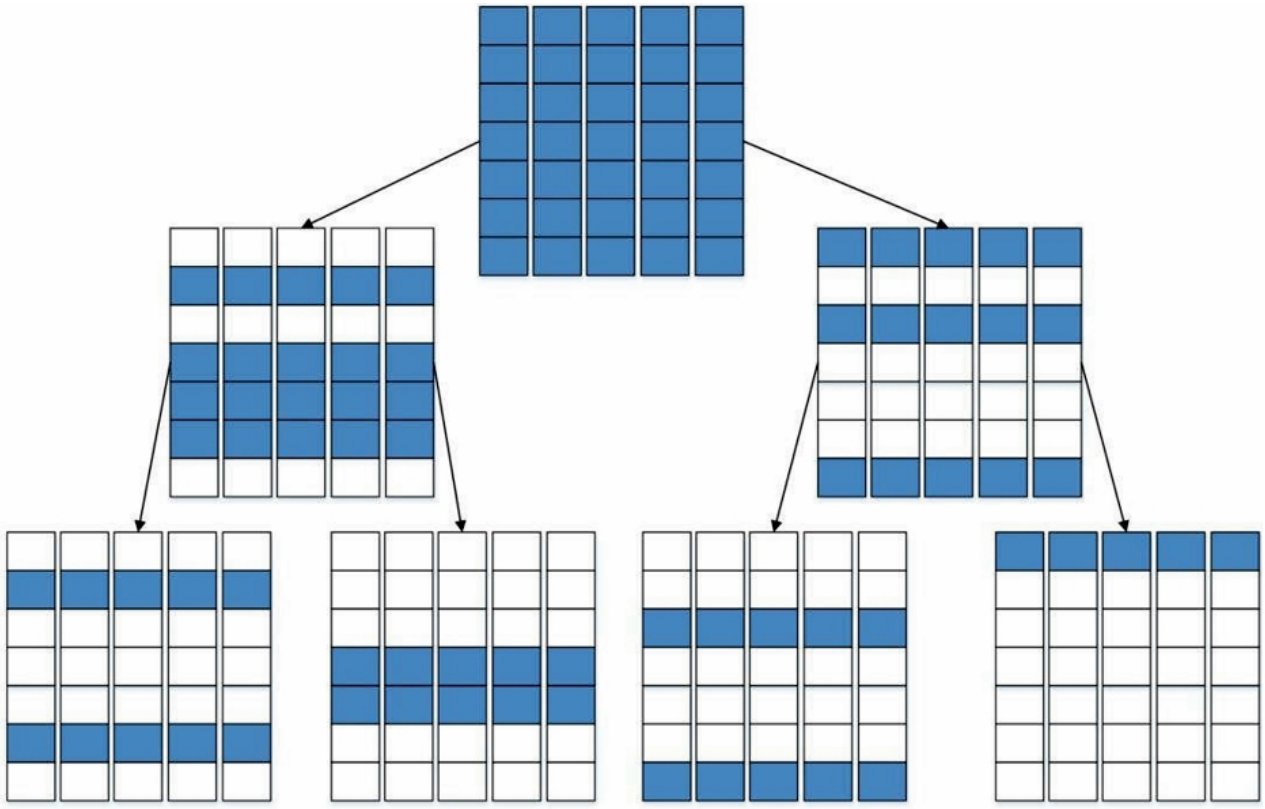
The task of the histogram is building gradients and Hessians for training samples at each boosting iteration. These can be summed into histogram bins according to the new, discrete features. Finding optimal splits for a decision tree is reduced to the simple problem of searching over histogram bins in a discrete space.

Bin-Matrix Layout

Let's start by choosing the memory layout for `bin-matrix` that gives the best performance. There are two choices:

- `column-major`
- `row-major`

To make the best choice, let's look at memory-access patterns for `bin-matrix`. First, we'll look at the `column-major` format (**Figure 2**).



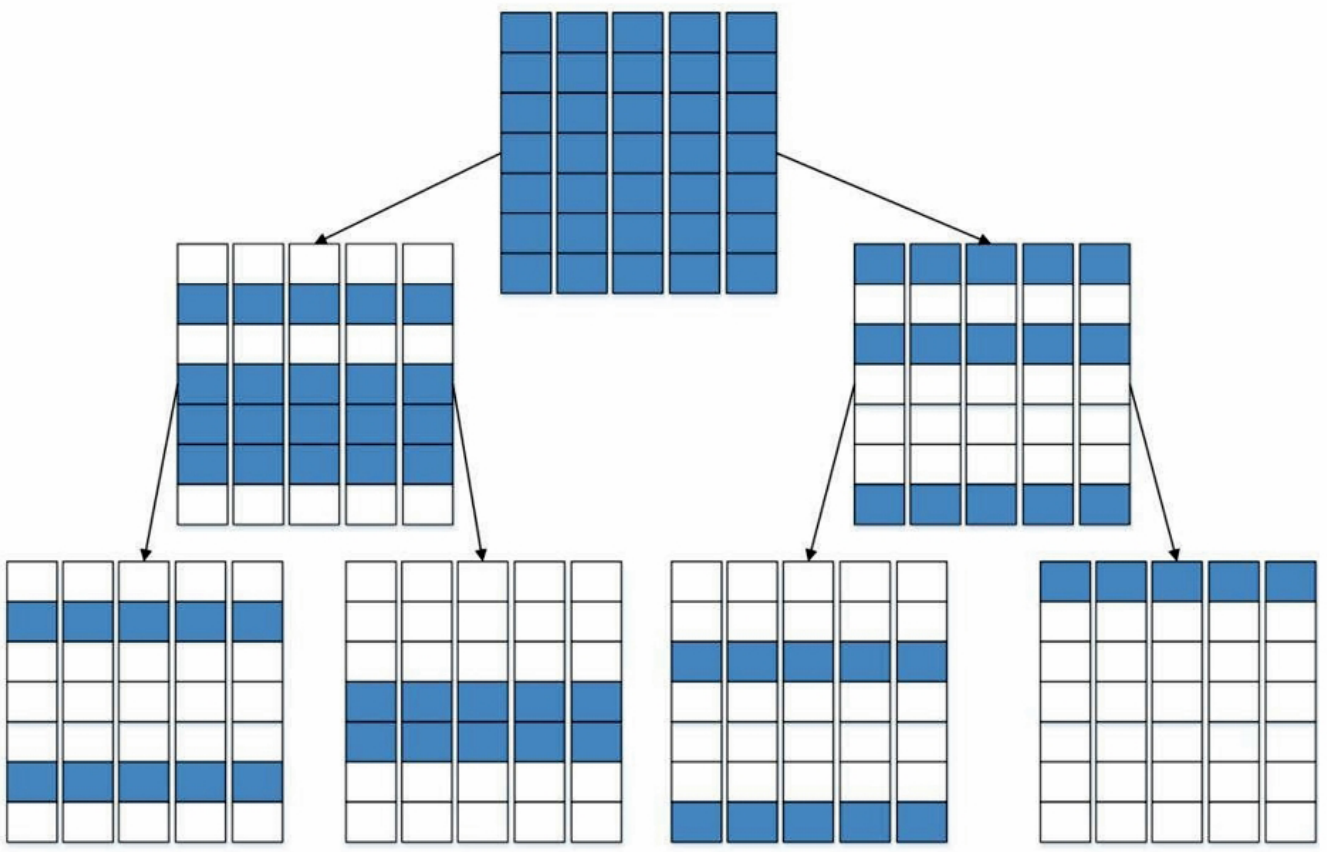
2 **`Column-major` format. Blue means this data is used for histogram computation. White means it is not.**

Before we can understand the advantages and disadvantages of memory layouts, we need to understand how memory works on the Intel Xeon processor. It fetches data to caches by cache lines (with a common length of 64 bytes). This means when we load one memory location, we'll actually fetch many others down the line. Efficient memory reading depends on how many numbers from the loaded cache line will be reused.

Now, let's look at disadvantages of the `column-major` approach:

- **For `not-root` nodes**, each cache line will touch only some of the elements needed for the current node for lower tree levels. This reduces the number of useful elements per cache.
- **If we read one column**, it might contain repeated bin indices. This means some data dependencies (read after write [RAW] dependency) will lead to poor hardware pipeline utilization.

Another approach is to use the `row-major` data format (**Figure 3**).



3 **The `row-major` data format**

There are three advantages to using the `row-major` data format over the `column-major` format:

1. **For `not-root` nodes**, memory access is still not uniform. But when we read a cache line, we fetch several adjacent elements that will be used for subsequent computations. This leads to better cache utilization and, as a result, better performance.
2. **There are no data dependencies** when we read one row.
3. **There is better load balancing for threading** due to the number of samples and threads.

This doesn't bring any performance issues when the histogram fits into the cache. But when the number of features is large enough, and the histogram does not fit into the cache, we can see performance degradation. In this case, it's better to use a mixed layout—`row-major` with some blocking by columns. However, usually histograms are not large enough and the `row-major` format shows better results because it works better with memory. All of these things lead us to choose the `row-major` approach in XGBoost and Intel DAAL.

Histogram Computation: Low-Level Optimizations

XGBoost versions 0.81 and lower had a simple way to build histograms:

```
for(int i = 0; i < n; i++)
    { for(int j = 0; j < p; j++) {
        int bin = bins[row_idx[i]*p + j];
        hist[bin].g += g[i];
        hist[bin].h += h[i];
    }
}
```

Let's consider how we can improve this.

Software Prefetching

In modern Intel® processors, built-in hardware prefetching can recognize many memory access patterns. Using explicit software prefetching doesn't improve performance in this case, and can even lead to a slowdown due to the overhead of prefetch instructions.

Threading

Histogram computation is easily parallelized by row blocks using a simple «`map`» pattern. As a result, we will have partial histograms on each thread. These should be merged to one before finding the best split.

In the previous implementation, we had the number of rows and a number of threads for each parallel task, with the smallest size being 8. A problem here is that some low decision tree nodes can contain only 10 to 500 samples. This is too small to parallelize efficiently because the overhead of creating the tasks can be larger than the cost of the histogram computation. To reduce this overhead and improve scalability, we limited block size to 512 rows.

Batch Operations

In the `depthwise` building mode, we can build nodes in parallel by levels in a tree. To enable parallelism—not only in building one node, but also by nodes—batch functions have been introduced. It allows us to create nested parallelism and improve scaling. **Table 4** shows the results after optimizing for histogram building. **Table 5** shows the changes in execution time by percentage.

Table 4. Performance before and after histogram building optimization

Data Set/Version	BuildHist XGB 1 (Before), s	BuildHist XGB 1 (After), s	Speedup
Higgs	82.2	7.1	11.6
MSRank	80.3	48.2	1.7
Mortgage	35.8	8.9	4.0

Table 5. Changes in the percentage of execution time

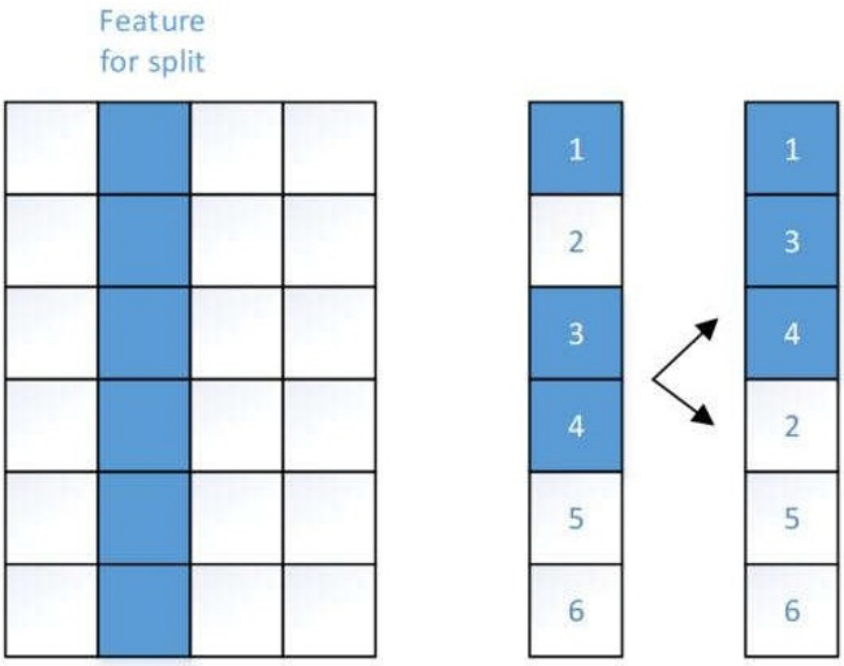
Function/ Data Set	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, %	4.9	13.5	17.4	57.4	1.0	5.7
MSRank, %	3.9	23.5	5.3	39.4	3.1	24.8
Mortgage, %	6.8	4.9	1.9	63.6	16.7	6.1

Before optimization, building histograms accounted for up to 82% of all training time. Now, it consumes less than 25%. Also, the new hotspot for all data sets is the `ApplySplit` function (which partitions a set of observations in some decision tree nodes to two parts). Let's look at how we might optimize this.

Partition Algorithm

The partition operation in the `ApplySplit` function (**Figure 4**) works like this:

- After finding the best split (we have an index of the feature to split), run `SplitIndex` and split the point in the current node to two successors. This is known as `SplitCondition`.
- Get the `SplitIndex` column.
- Choose the elements in the column that are less than `SplitCondition`. Indices of these elements should go to one part. The rest go into another part.



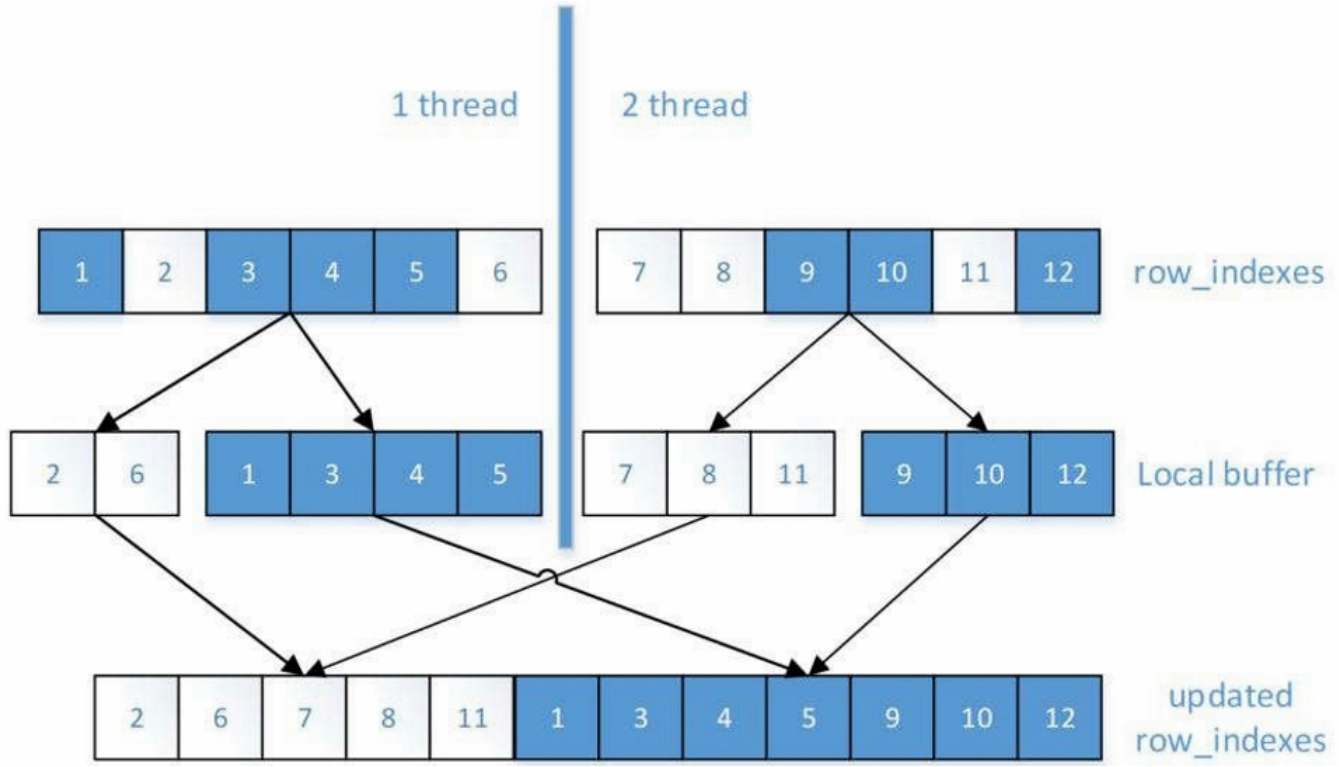
4 Partition algorithm

As a result, the first indices in the row will represent indices of observation, which should go to the left successor. The second part of the row goes to the right successor. According to the algorithm, this partition should be stable. This means that indices in each part should be sorted. But there are two problems:

1. `Bin-matrix` is `row-major`, but we need to read only one column. This leads to reading with constant stride.
2. Threading requires communication between threads.

The main problem with this kernel is finding opportunities for efficient parallelization. Here's the scheme we used (**Figure 5**):

1. Divide row index array among threads.
2. Make a local partition in some local buffer.
3. Synchronize the threads with a goal of sharing the number of elements in each local partition.
4. Update the row index by copying elements from local buffers.



5 **Parallelization scheme**

Table 6 shows our results after partition optimizations. **Table 7** shows changes in the percentage of execution time.

With these optimizations in place, new performance hotspots emerge:

- **`EvaluateSplit`:** Find the best split by histograms.
- **`UpdatePredictionCache`:** Update predictions for the training dataset according to the built tree.
- **`InitData`:** Choosing observations for training, among other things.

Let's try to reduce the execution time for these, too.

Table 6. Results after partition optimizations

Data Set/Version	ApplySplit XGB 1 (Before), s	ApplySplit XGB 1 (After), s	Speedup
Higgs	30	4.1	7.4
MSRank	81	9.9	8.1
Mortgage	117	4.8	24.1

Table 7. Changes in the percentage of execution time

Dataset/ Function	InitData	BuildHist	Evaluate-Split	ApplySplit	Pre-Processing	UpdatePre-dictionCache
Higgs, %	9.8	26.8	34.6	15.4	2.1	11.3
MSRank, %	5.9	36.0	8.1	7.4	4.7	37.9
Mortgage %	17.4	12.5	4.9	6.8	42.8	15.6

Other Optimizations

Finding the Best Split

This kernel decides what feature and split point to choose for node splitting. Here are two optimizations that have been done:

1. **To handle missing values**, XGBoost tries to find the best split by two searches. In the first, all missing values come to the left node. In the second, they come to the the right node. In theory, this should improve performance twofold.
2. **Compute the best split for both left and right nodes** on one thread to keep and reuse the data that is already in the caches.

.UpdatePredictionCache and InitData

The main performance problem with `UpdatePredictionCache` and `InitData` was simply sequential code that could easily be threaded. Adding OpenMP reduced the time for these functions significantly.

Tables 8, **9**, and **10** show the breakdown by kernel after all optimizations.

Table 8. Breakdown for XGBoost 0.81

Data Set/ Function	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, s	3	219	9	30	1	3
MSRank, s	8	673	11	81	6	51
Mortgage, s	12	101	3	117	31	11

Table 9. Breakdown for XGBoost 1.0

Data Set/ Function	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, s	0.1	7.1	3	4.1	0.4	1.0
MSRank, s	0.4	48.2	10	9.9	4	49
Mortgage, s	0.2	8.9	0.3	4.8	4	0.5

Table 10. Final breakdown in percentages

Data Set/ Function	InitData	BuildHist	EvaluateSplit	ApplySplit	PreProcessing	UpdatePre-dictionCache
Higgs, %	0.8	43.7	21.5	25.1	2.7	6.2
MSRank, %	0.3	39.6	7.9	8.1	3.5	40.6
Mortgage, %	1.2	47.6	1.6	25.8	21.1	2.8

To-Do List

There are more changes we can make to achieve even better performance on Intel Xeon processors:

- Change the type of `bin-matrix` from `int32` to `uint8` to reduce the amount of memory needed for histogram computation.
- Introduce a cache for prediction in multiclass cases.
- Use SIMD code to build histograms and partitions.

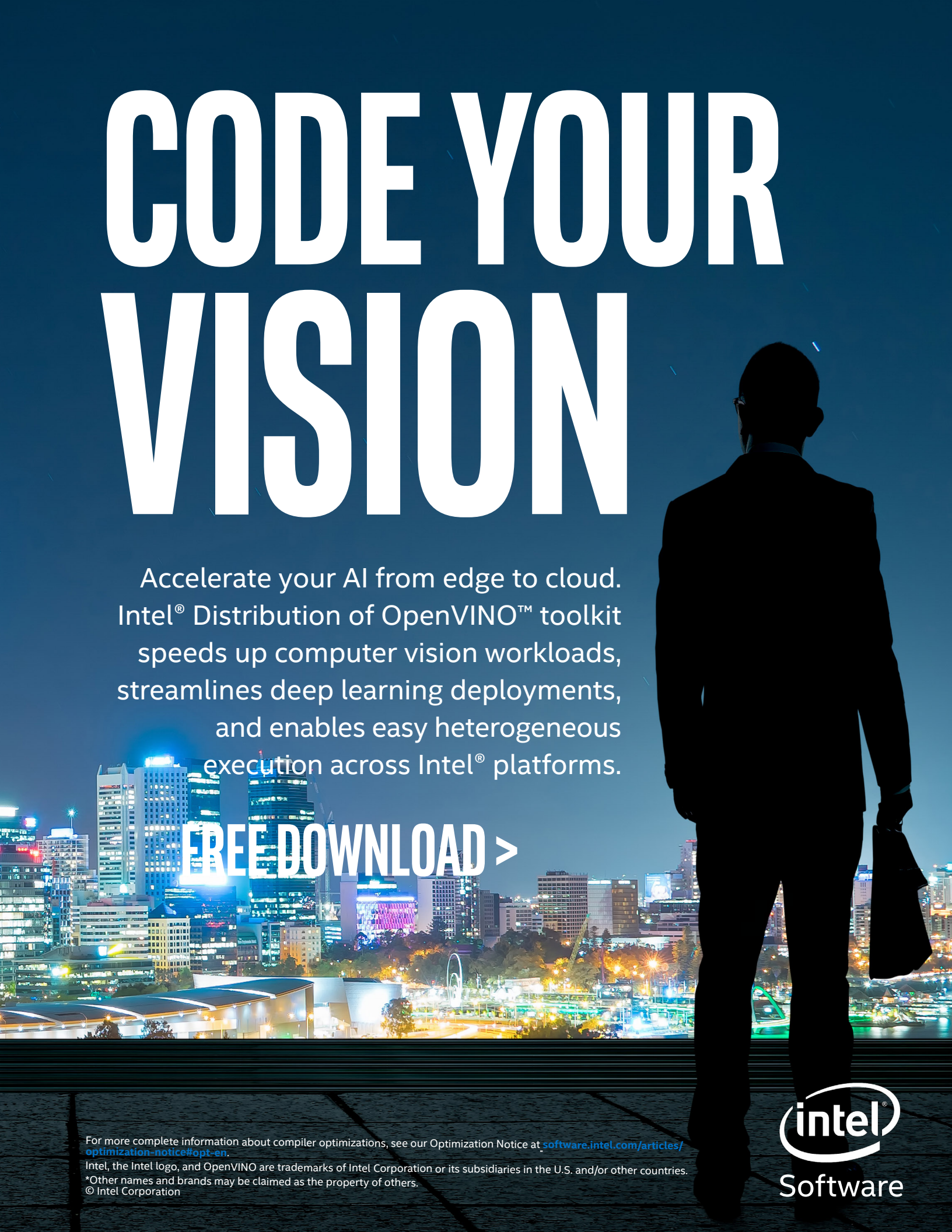
These optimizations will have to wait for a future article.

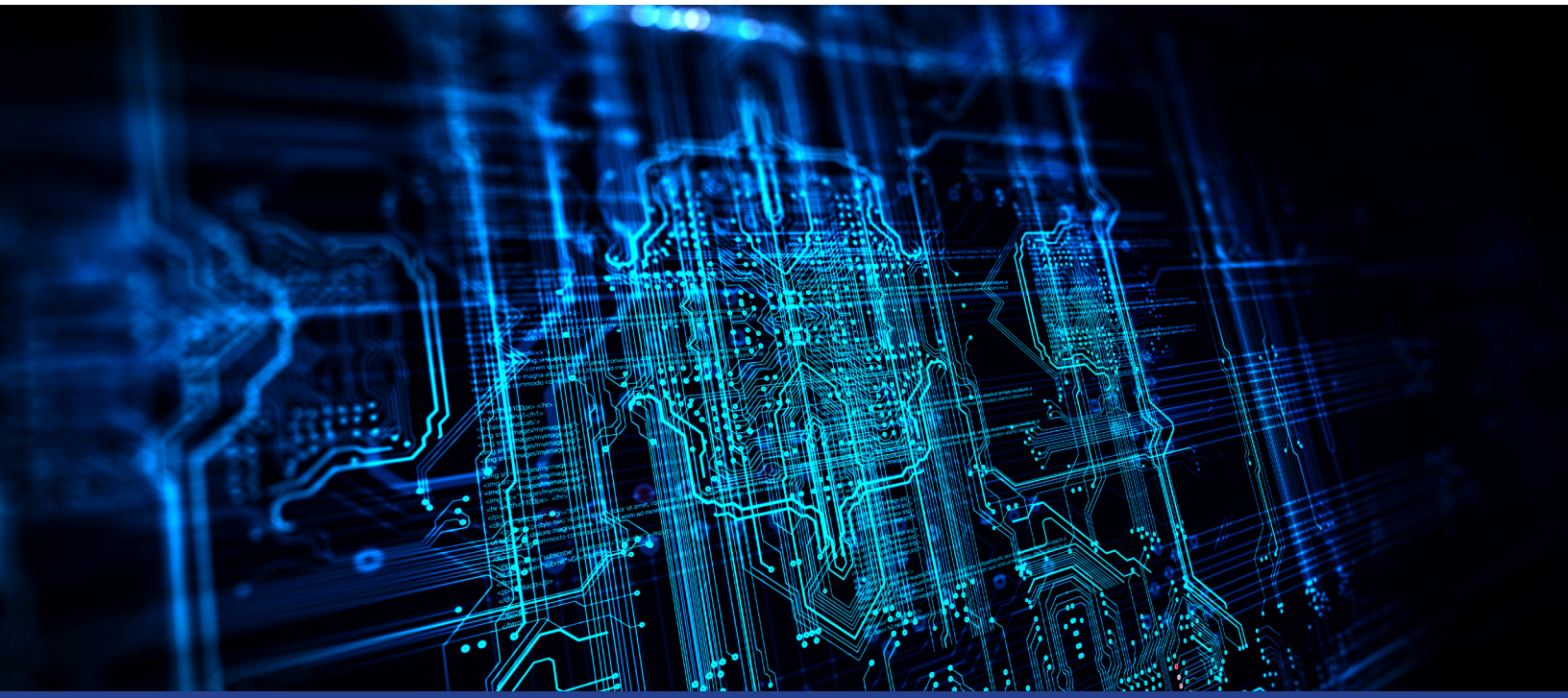
References

1. Tianqi Chen and Carlos Guestrin. **XGBoost: A Scalable Tree Boosting System**. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016.
2. Hyunsu Cho. Speeding Up Gradient Boosting for Training and Prediction.
3. **Intel DAAL**

Appendix: Gradient Boosting Training Configuration

- **CPU configuration:** c5.metal AWS instance (2nd generation Intel® Xeon® Scalable processors, two sockets, HT:on, Turbo:on, OS: Ubuntu 18.04.2 LTS, total memory of 193 GB (12 slots/16GB/2933 MHz), BIOS: 1.0 Amazone EC2 (ucode: 0x5000017), OMP environment: OMP_NUM_THREADS=48 OMP_PLACES={0}:96:1).
- **GPU configuration:** p3.2xlarge AWS Instance (CPU: Intel® Xeon® E5-2686 v4 processor @ 2.30GHz, one socket, four cores, HT:on, Turbo:on, GPU: Tesla* V100-SXM2-16G (driver version: 410.104, CUDA* version: 10.0), OS: Ubuntu* 18.04.2 LTS, total memory: 61 GB (4 / 13312 MB), BIOS: 4.2 Amazone* EC2 (ucode: 0xb000037)).
- **Software:** XGBoost* 0.81: download from PIP*. XGBoost 1.0: master (ef9af33a000f09dbc5c6b09aee133e38a6d2e1ff), compiler – G++ 7.4, nvcc 9.1. Intel DAAL: 2019.4 version, downloaded from Conda*. Python* environment: Python 3.6, Numpy 1.16.4, Pandas* 0.25, Scikit-learn* 0.21.2.


CODE YOUR VISION
Accelerate your AI from edge to cloud. Intel® Distribution of OpenVINO™ toolkit speeds up computer vision workloads, streamlines deep learning deployments, and enables easy heterogeneous execution across Intel® platforms.
FREE DOWNLOAD >
For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel, the Intel logo, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation
intel
Software



DETECTING AND MITIGATING FALSE SHARING IN MULTI-PROCESSORS

Get Big Performance Benefits for Your Multithreaded Applications

Ramesh Peri, Senior Principal Engineer, Intel Corporation

Multi-core, single-node systems with multiple layers of local caches per core are now commonplace. An average laptop has at least two or more cores, while servers have tens of cores. Software takes advantage of these cores through multithreading. Since the threads are operating on shared data structures in a common address space, copies of data present in local caches of different cores must coordinate with each other. For example, if two threads are accessing the same memory location, the memory subsystem needs to validate and update the copies present in local caches as threads update them. (This is independent of the synchronization mechanisms that multithreaded programs use to ensure atomic updates to shared memory locations.) The memory subsystem guarantees this

coherence by implementing a complex cache coherence protocol, which can impact performance when multiple threads access and update shared memory locations.

This article explores the cache coherence protocols and their impact on the performance of multithreaded programs.

Cache and Cache Coherency Protocols

A cache is a small, fast memory the CPU uses to store copies of frequently used locations from main memory. In a multi-core system, every CPU has its own cache that stores copies of main memory locations. To maintain program correctness, the CPUs synchronize and update the contents of these caches during execution. Data is transferred from memory to the caches, typically in blocks of 64 or 128 bytes. When the CPU needs to read or write a location in memory, it first checks whether the address is present in any of the cache lines. If it's present, the CPU fetches it from there. If it's not present, a new entry is allocated in the cache for future use.

In the case of multi-core machines, the CPU needs to check its local cache, as well as the caches of the other cores, before it decides whether to bring the contents from memory. To reduce the amount of checking needed on every memory access across all caches of all cores, certain information is stored in every cache line. This information is known as MESI (modified/exclusive/shared/invalid).

On the first load of a cache line, the CPU marks the cache line as "Exclusive" access. All subsequent loads can use this data as long as it stays in Exclusive mode. If the same cache line is loaded by another CPU, then the cache line is marked as "Shared" in all caches of other CPUs. If a CPU writes to a cache line marked as "Shared," then it's marked as "Modified" and all other CPUs are sent an "Invalid" cache line message. If the CPU sees a cache line marked "Modified" being accessed by another CPU, it stores the cache line back to memory and marks its cache line as "Shared." The other processor that's accessing the same cache line incurs a cache miss and fetches the updated copy from main memory.

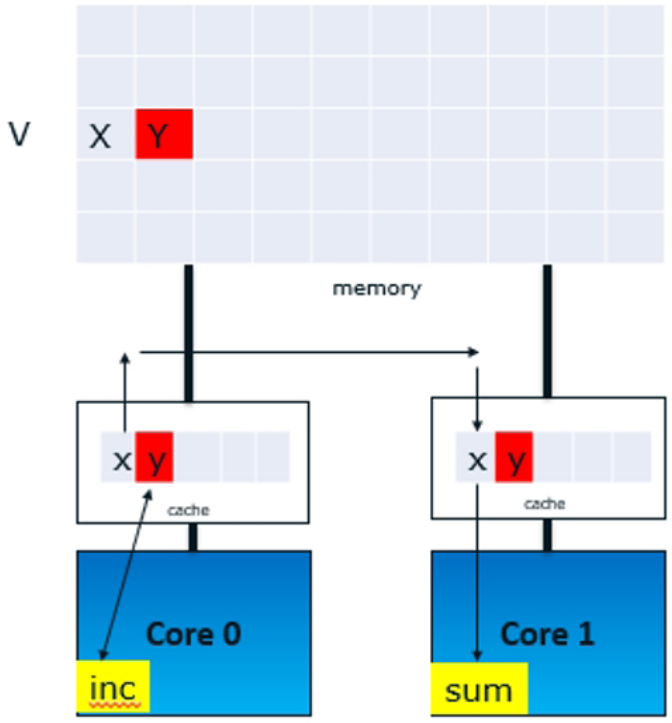
We can see that accessing a memory location by a CPU in a "Modified" state in another CPU cache incurs:

- A write to memory
- A read from memory

For memory locations accessed by multiple threads running on different cores, this is necessary to provide up-to-date values and maintain guarantees of the memory model.

For efficiency, the cache coherence protocol operates at the level of cache lines instead of individual bytes. This gives rise to a phenomenon called false sharing (**Figure 1**). In false sharing, threads

```
struct {
   int x;
   int y;
} v;
/* sum & inc run in parallel */
int sum(void)
{
   int i, s = 0;
   for (i = 0; i < 1000000; ++i}
      s+=v.x;
   return s;
}
void inc(void)
{
   int i;
   for (i=0; <`1000000; ++i)
   v.y++;
}
```



1 **False sharing in a multi-core system**

accessing different memory locations that happen to be present on the same cache line also go through coherence mechanisms of the cache coherence protocol, incurring performance penalties. In some cases, this performance penalty can be significant.

In **Figure 1**, the threads `sum` and `inc`, running on core 0 and core 1, respectively, access two different memory locations that happen to be in the same cache line. Since the cache coherency protocol operates at cache line granularity, every access to `x` by the sum thread after an access to `y` by the `inc` thread will incur a performance penalty as though the two threads are accessing same variable.

Detecting Sharing Among Caches

The performance monitoring unit on Intel® processors has the ability to monitor different kinds of events that happen in the microarchitecture during the execution of a program. There are a number of events related to referencing a memory location in a modified, exclusive, or shared state in the cache of some other CPU in the system. These events include the string `'HIT<M/E/S>'`. By monitoring these events, we can see if there's any cross-referencing of cache lines among cores. For example, on **Goldmont**-based platforms, we can monitor `MEM_UOPS_RETIRED.HITM` to determine the number of references to cache lines that are in the modified state in other CPUs' caches. In the case of **Skylake** family processors, we can monitor false sharing by looking

at a number of events and applying a formula over them. (**Intel® VTune™ Amplifier** computes all the needed metrics automatically.)

A Microbenchmark to Illustrate False Sharing

The code in **Figure 2** creates a specified number of threads and gets those threads to access an array with a specified pattern. This code takes as arguments the number of threads and the starting index for each of these threads to access an array. Running this code with arguments `4 0 1 2 3` generates the access pattern shown in **Figure 2**. Here, we can see that all four threads are operating on different array elements located in the same 64-byte cache line (**Figure 3**). This access pattern causes the processor to "ping-pong" the cache line among the local caches of various cores running these threads, which can limit parallel performance.

Running the code with arguments `4 0 64 128 192` results in the access pattern shown in **Figure 4**. Each row corresponds to a different cache line. We can see that every thread is operating on array elements that are in different cache lines.

It's important to note that in both instances (**Figures 3** and **4**), the number of instructions retired, and the memory loads and stores, are exactly the same for every thread. Also, all these threads are always accessing different elements in the array. So we can conclude that any performance difference between these two runs of the program is due to the behavior of the cache subsystem.

Performance Evaluation of the False Sharing Microbenchmark

The microbenchmark was run in the Intel VTune Amplifier performance analysis tool for cases with and without false sharing. **Figure 5** shows the summary view with false sharing and **Figure 6** shows the summary view without false sharing. We can see that in both cases, the amount of work done by both the runs is the same (since both runs are executing the same number of instructions, which is around 5 billion). But the runs take different numbers of clock cycles—about 12.4 billion clocks with false sharing and 3.5 billion clocks without false sharing, which is almost 3x slower. It's important to note that Intel VTune Amplifier detects and reports this as being 100% store-bound, because the threads are waiting for the stores of other threads to complete, which is caused by false sharing of cache lines.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// shared array accessed by all threads
int a[256] __attribute__ ((aligned(64)));

/* this function is run by the second thread */
void *thread_func(void *vptr)
{
    int indx = *((int *)vptr);
    for (int j=0;j<1000;j++)
       for (int k=0;k<10000;k++)
          for (int i=indx;i<(indx+32);i+=4)
             a[i]=a[i]+1;
    return NULL;
}

void usage(char *name)
{
    printf("USAGE: %s <n> <offset-1> ... <offset-n>\n",name);
    exit(0);
}

int main(int argc, char **argv)
{
    int num_threads;
    pthread_t *threads;
    int *access_pattern;
    if (argc < 2) usage(argv[0]);
    num_threads = atoi(argv[1]);
    if (argc != (num_threads+2)) usage(argv[0]);
    // allocate space for threadids and the access pattern
    access_pattern = (int *)calloc(num_threads,sizeof(int));
    threads = (pthread_t *)calloc(num_threads, sizeof(pthread_t));
    for (int i=0;i<num_threads;i++)
       access_pattern[i]=atoi(argv[i+2]);
    printf("Running %d threads with access pattern: ",num_threads);
    for (int i=0;i<num_threads;i++)
       printf("%d ", access_pattern[i]);
    printf("\n");
    // create threads
    for (int i=0;i<num_threads;i++)
       if (pthread_create(&threads[i], NULL, thread_func, &access_pattern[i])) {
          fprintf(stderr, "Error creating thread %d\n",i);
          return(1);
       }
    // wait for all threads to finish
    for (int i=0;i<num_threads;i++)
       if (pthread_join(threads[i], NULL)) {
          fprintf(stderr, "Error joining thread %d\n",i);
          return(2);
       }
    return 0;
}
```

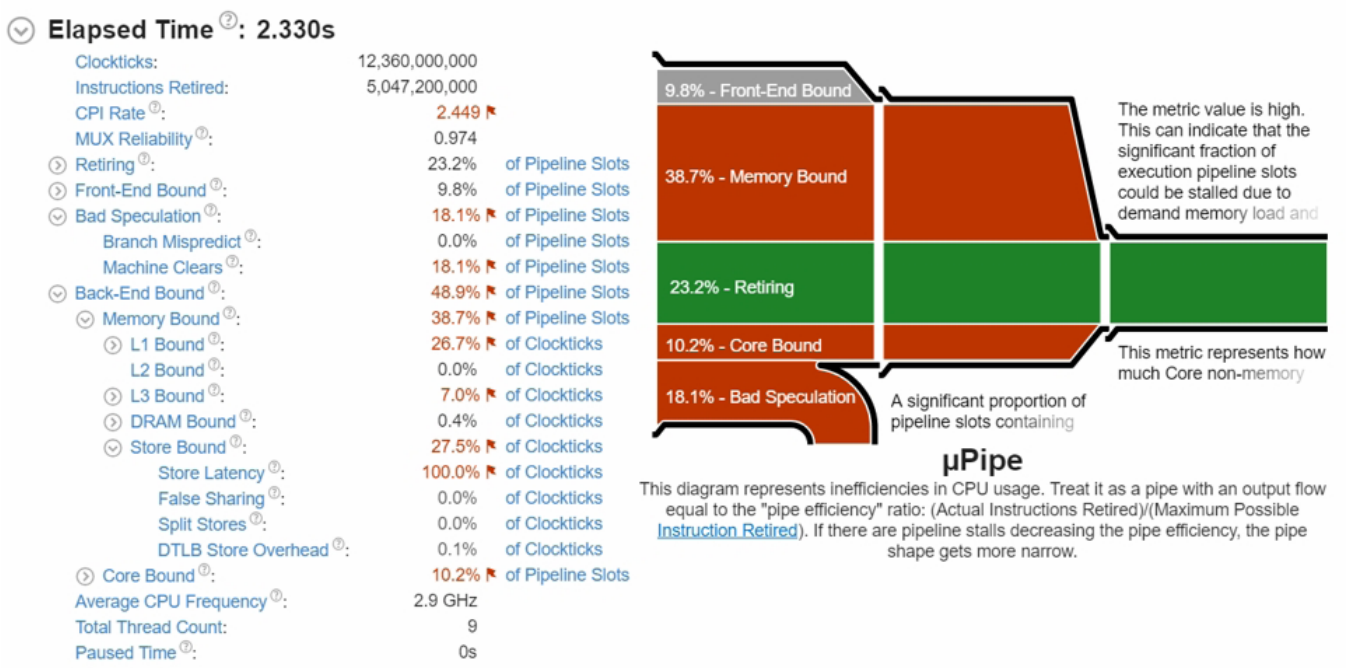
2 **Microbenchmark for generating different memory access patterns**

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	....
																...
																...
																...

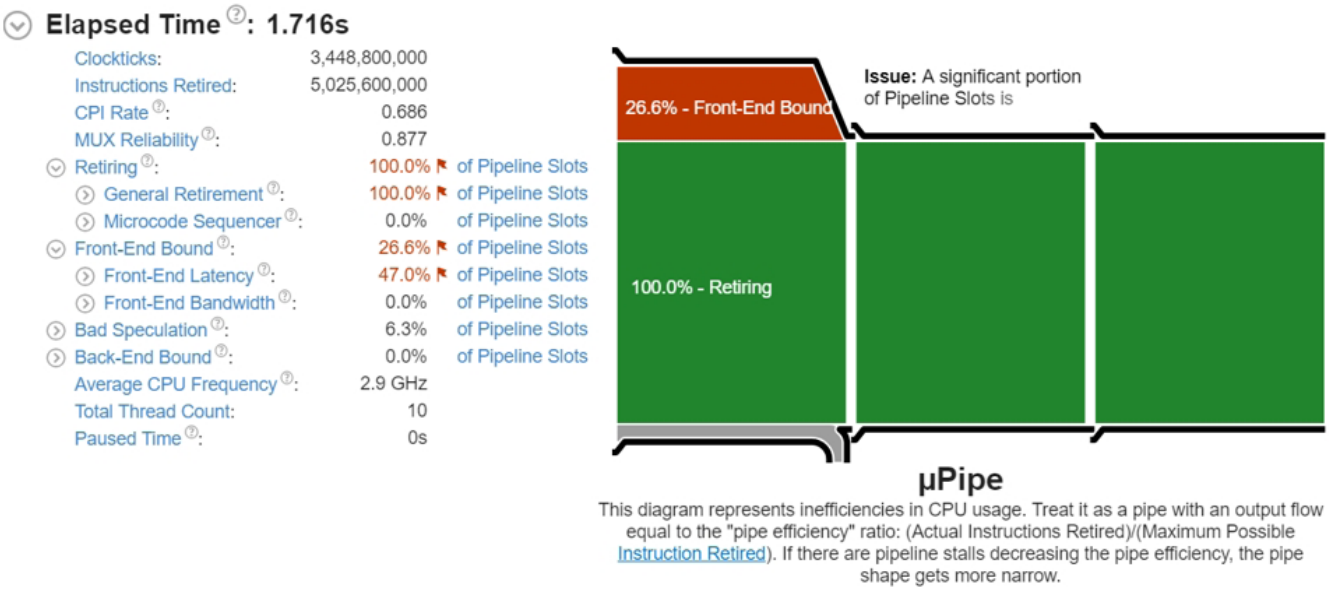
3 **Memory access pattern generated by the code in Figure 2 with arguments `4 0 1 2 3`, which has false sharing for caches with 64-byte cache lines.**

1				1				1				1				1				...
	2				2				2				2				2			...
		3				3				3				3				3		...
			4				4				4				4				4	...

4 **Memory access pattern generated by the code in Figure 2 with arguments `4 0 64 128 192`, which does not exhibit false sharing for caches with 64-byte cache lines.**



5 **Summary view from Intel VTune Amplifier for the memory access pattern that exhibits false sharing (Figure 3)**



6 **Summary view of Intel VTune Amplifier for memory access pattern that does not exhibit false sharing (Figure 4)**

Preventing False Sharing

False sharing can have a big performance impact in multithreaded applications and can arise from the way the compiler, or the user, lays out shared data structures. Performance for the microbenchmark can be 3x worse due to false sharing. Profiling tools like Intel VTune Amplifier have built-in analysis tools that can detect false sharing and alert the developer.

For More Information

1. **MESI Protocol**
2. **Intel VTune Amplifier**
3. **Intel® 64 and IA-32 Architectures Software Developer Manuals**

SPEEDING UP SIMULATION ANALYSIS WITH YT* AND INTEL® DISTRIBUTION FOR PYTHON*

How to Boost Analytics Performance on Intel® Xeon® Scalable Processors

Salvatore Cielo, PhD, Scientific Computing Expert, Leibniz Supercomputing Centre; Luigi Iapichino, PhD, Scientific Computing Expert, Leibniz Supercomputing Centre; Fabio Baruffa, PhD, Technical Consulting Engineer, Intel Corporation

As modern scientific simulations grow in size and complexity, even their analysis and post-processing becomes increasingly demanding—calling for the use of HPC resources and methods. `yt`* is a parallel, open-source, post-processing Python* package for numerical simulations in astrophysics, made popular by its:

- **Cross-format** compatibility
- **Active community** of developers
- **Integration** with several other professional Python instruments

Intel® Distribution for Python* enhances `yt`'s performance and scalability by optimizing lower-level libraries like NumPy* and SciPy*, which make use of the optimized **Intel® Math Kernel Library (Intel® MKL)** and the **Intel® MPI Library** for distributed computing. The library package **yt** is used for analysis tasks like:

- Integration of derived quantities
- Volumetric rendering
- 2D phase plots
- Cosmological halo analysis
- Production of synthetic X-ray observation

In this article, we'll provide a brief tutorial for installing `yt` and Intel Distribution for Python and show how to execute each analysis task. Compared to the Anaconda* Python distribution, this solution can deliver net speedups up to 4.6x on **Intel® Xeon® Scalable processors**.

Installing `yt` in the Intel and Anaconda* Python Environment

We'll need a Python 3 environment (we chose version 3.6) to create the Anaconda package manager. For the installation of the Intel-optimized software, we need to add the Intel channel first:[1]

```
conda config --add channels intel
```

Now we can create the environment, which we name `my_yt`, and install `yt` and all the packages we'll need, including the required dependencies. For Intel, the procedure is:

```
conda create -y -c intel -n my_yt python=3.6
source activate my_yt
conda install -y -c intel numpy scipy sympy mpi4py matplotlib
conda install -y -c conda-forge yt
conda install -y -c jzuhone -c astropy pyxsim
```

Repeating the channel name ensures the correct origin of all dependency packages, while the `-y` option suppresses the confirmation prompt. Conda's installation help can be displayed at any time with `conda install -h`.

To use Anaconda Python, which we need for performance comparisons, just replace `-c intel` with `-c anaconda` (which changes the environment name). Alternatively, you can use the all-in-one installation script from the `yt` webpage, providing the Anaconda version within the Miniconda* environment.

Tutorial: Common Post-Processing Tasks

Now let's review a number of tasks that allow us to analyze simulation data. (You can see these, and more advanced options, in the `yt` 3.5 tutorial.[2]) We'll use a cosmological simulation including stars, dark matter, and interstellar gas run with ENZO.*[3] Since `yt` reads several formats, it's a good general-purpose volume renderer.

To ensure proper `mpi4py` parallelization, the tasks must be scripted and Python invoked in parallel, with `mpiexec` or equivalent (e.g., `mpiexec -np 8 python my_tasks.py`). We begin by importing `yt` and `mpi4py`, enabling `mpi4py`, loading the chosen snapshot, and performing a spherical selection (sp) of of

```
import yt, mpi4py
yt.enable_parallelism()
ds = yt.load("RD0028/RedshiftOutput0028")  # Opening file header only
sp = ds.sphere("c", (10.,"Mpc"))           # Central 10 Megaparsec sphere
```

`yt` knows several derived quantities to compute from the snapshot data fields. We choose the total angular momentum per unit mass `j` of both gas and particles (dark matter, stars), which is interesting for halo shape studies or matter accretion purposes (e.g., around black holes). This is calculated algebraically from 3D positions and velocities, and integrated over `sp` in just one line:

```
j  = sp.quantities.angular_momentum_vector(use_gas=True, use_particles=True)
```

Then we use `print(j)` to print the value of `j` in the correct unit system.

Likewise, it's easy to learn about the thermodynamic state of the gas through a `phase-plot`, binning the fields in 2D histograms. In this way, you can get pressure-volume diagrams for energy measures or density-temperature diagrams showing the gas' equation of state.

```
pp = yt.PhasePlot(sp, "density", "temperature", ["cell_mass"], weight_field=None)
pp.save()
```

The two simply additive tasks described above make use of `yt`'s efficient grid parallelism (i.e., a straightforward distribution among the active `mpi4py` processes of the basic elements—grids or particles—over which the fields are defined).[4] For more complex tasks, like volume rendering or cosmological halo analysis, we need to use a proper spatial decomposition of the domain (more powerful but less efficient). Volume renderings in `yt` are obtained through ray casting. In the most compact instance, an image `im` of the gas density is printed by:

```
im, sc = yt.volume_render(ds,  ('gas', 'density'), fname='volume.png')
```

Refined controls on the scene object `sc` (transfer function, colormap, camera), and over the image itself (sigma clipping, opacity), are available.

Finally, let's look at synthetic X-ray observations with the `pyXSIM` package, using `SOXS` (dependency of `pyXSIM`) to simulate a real telescope (Chandra's ACIS-I*).[5] We begin by setting the parameters of the telescope and observation, including a thermal X-ray emission model for the gas, telescope collecting area, and the exposure time:

```
import soxs, pyxsim
soxs.soxs_cfg.set("soxs", "response_path", "./soxs_responses" )
redshift = 0.05
src_model = pyxsim.ThermalSourceModel("apec", 0.05, 11.0, 10000, Zmet=0.3)
exp_time = (100., "ks")
area     = (2000.0, "cm**2")
sp = ds.sphere("c", (50.,"Mpc")) # Enlarge to 50 Megaparsec
```

`pyxsim` then computes individual photon packages using Monte Carlo radiative transfer (`photons task`), and finally projects them onto the detector (`events task`). The model includes hydrogen absorption.

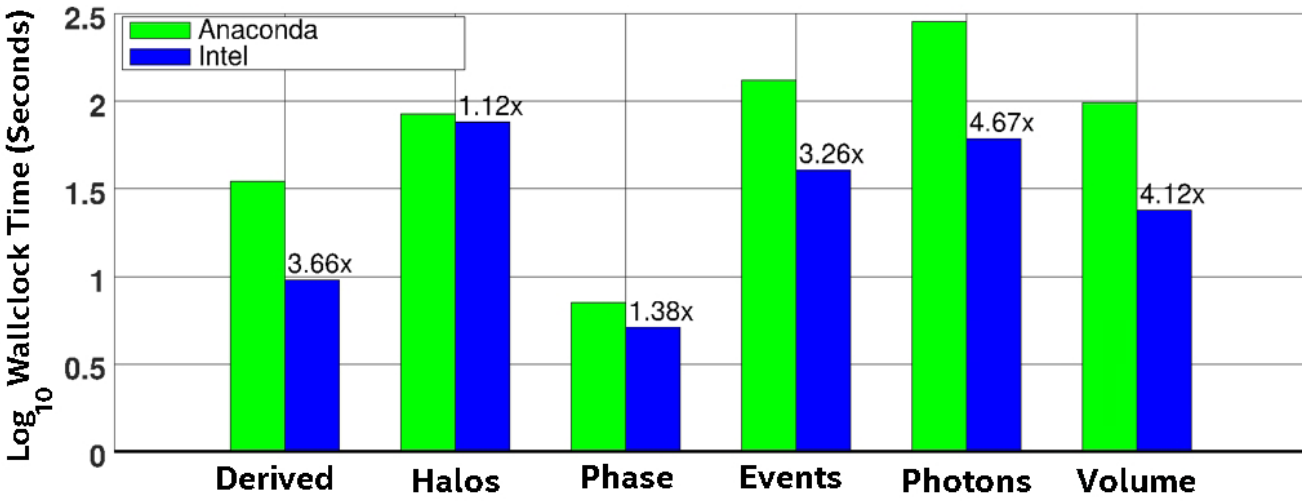
```
# Computation 1/2: Montecarlo radiative transfer
photons = pyxsim.PhotonList.from_data_source(sp, redshift, area, exp_time, src_model)
# Computation 2/2: Photons produce events into simulated detector
events_z = photons.project_photons("z", (45.,30.), absorb_model="tbabs", nH=0.04)
```

The data can then be printed out, and the simulated instrument applied (optional). You can also adjust exposure, celestial coordinates (here, 45 and 30 degrees), and photon energy extrema (here, set to 0.5 and 11 `electronVolt`). This completes all scripted tasks.

```
events_z.write_simput_file("RD0028", overwrite=True) # Warning: very large fits file!
soxs.instrument_simulator("RD0028_simput.fits", "evt.fits", (100.0, "ks"), "acisi_cy0",
[45., 30.], overwrite=True)
soxs.write_image("evt.fits", "img.fits", emin=0.5, emax=11.0, overwrite=True)
exit()
```

Speedup from Intel Distribution for Python

We ran a scaling test of all the tasks for both Anaconda and Intel Distribution for Python over a single Intel Xeon Scalable processor node, and plotted the shortest execution time (median of 20 measurements) in **Figure 1**.



1 **Performance comparison of Anaconda and Intel Distribution for Python on all tasks. We plot the shortest execution time (log scale) on one Intel Xeon Scalable processor node. Intel Distribution for Python improves the performance up to 4.6x.**

Intel Distribution for Python improves performance up to a factor 4.6x, with longer tasks tending to show larger improvements. The halos task is an exception, since instead of grid or spatial parallelism, `yt` distributes individual halos (once they're found) among the processes. The full data fields are still accessed for computation, so the task is not embarrassingly parallel, but the work sharing is easier. The speedup we see is due to better scaling, and the value may increase with the number of halos.

Concerning code scalability, Anaconda always performed better in serial than in parallel, except for the events task, scaling up to two `mpi4py` processes. So the only convenient parallelization on Anaconda is, unfortunately, `yt`'s embarrassingly parallel scheme over time series or different objects.[4] Intel Distribution for Python scales easily up to 8 or 16 cores, allowing a much better usage of the shared resources—which is important for larger tasks or simulations.

References

1. **Installing Intel® Distribution for Python* and Intel® Performance Libraries with Anaconda***
2. **The yt Project Overview**
3. **ENZO: An Adaptive Mesh Refinement Code for Astrophysics**
4. **Parallel Computation with yt**
5. **pyXSIM Documentation**

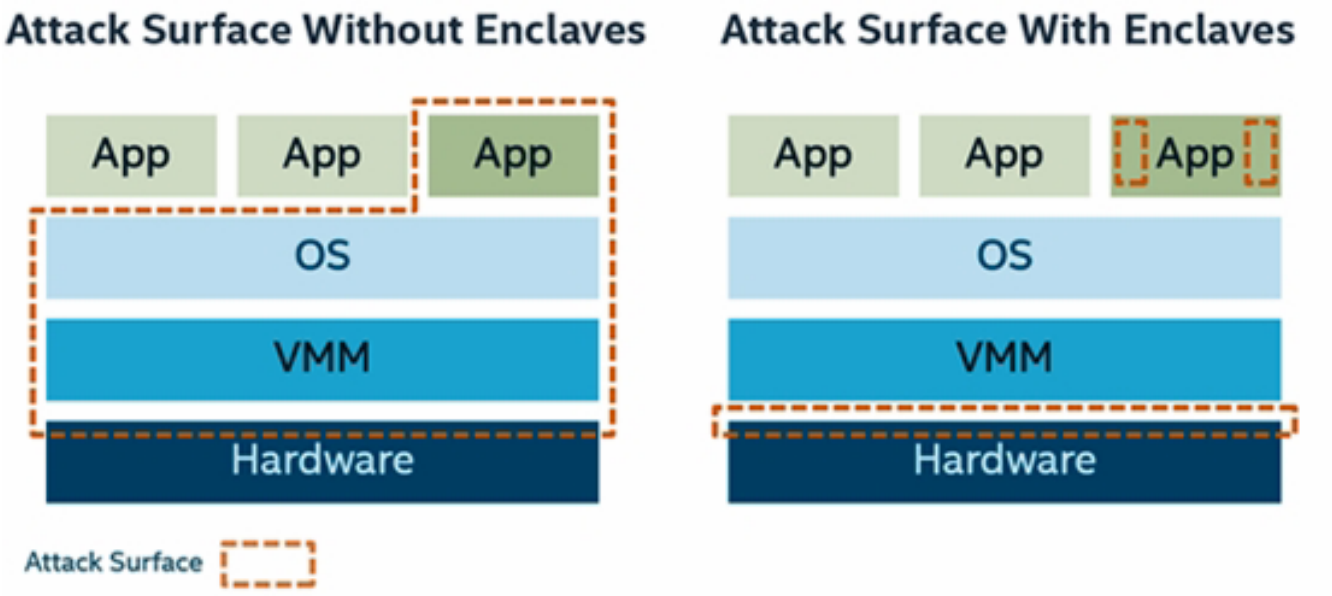
INTEL® SOFTWARE GUARD EXTENSIONS

Using Hardware-Based Isolation and Memory Encryption to Provide More Code Protection in Your Applications

Rama Kishan Malladi, Performance Modeling Engineer, Intel Corporation

Intel® Software Guard Extensions (Intel® SGX) is a set of CPU instructions that help application developers protect their code and data regions from disclosure and/or modification. The data being protected could include sensitive information such as passwords, account numbers, financial information, or health records that are intended to be accessed only by a designated recipient.

Intel SGX enables applications to create protected enclaves in an application's address space. These enclaves are built into, and loaded as, a Windows* Dynamic Link Library (DLL) file. Using enclaves, Intel SGX helps reduce the surface of an attack, as shown in **Figure 1**.



1 **Vulnerability without and with Intel® SGX enclaves**

Features

To design an application using Intel SGX, you split the application into two components:

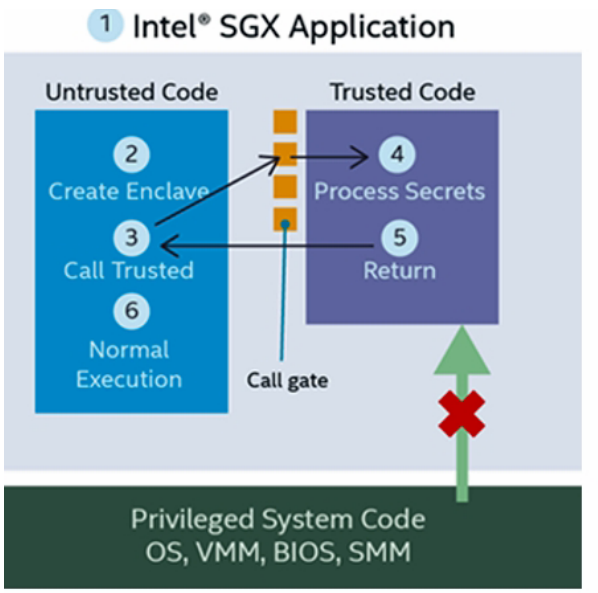
1. **A trusted region**
2. **An untrusted region**

The trusted region (code and data) constitutes an enclave. An application can have one or more enclaves. The untrusted code is the rest of the application. It's a good idea to keep the trusted region of code and data as small as possible.

Figure 2 shows how to build an app with both trusted and untrusted regions. The trusted regions (enclaves) are placed in trusted memory. Enclaves are invoked, executed, and returned. External access to enclave code/data is denied.

Intel SGX provides several protections from attacks:

- **Enclave memory is protected.** It can't be read/written from outside the enclave.
- **Enclave memory is encrypted.** It can only be decrypted by the CPU (with a stored key).
- **A production enclave can't be debugged by a hardware or software debugger.** A debug build and Intel SGX debugger are required for debugging.
- **Enclave functions can't be entered through function calls/jumps.** SGX instructions are required. The entry points into an enclave are predefined during compilation.
- **Intel SGX protects the confidentiality and integrity** of the enclave code and data.



2 **Intel® SGX application flow**

Note that to use Intel SGX, the processor must support it. Also, Intel SGX must be enabled in the BIOS and the software stack (Intel® SGX Platform Software) must be installed. (Check references 1 and 2 for details.)

Instructions and Structures

Intel SGX is a collection of two instruction extensions:

1. **SGX1 instantiates a protected container**, referred to as an enclave.
2. **SGX2 allows additional flexibility** in the runtime management of enclave resources and thread execution within an enclave.

The Intel SGX instructions are grouped under Ring 0 and Ring 3 privilege.

Table 1 shows the Intel SGX1 instructions for enclave setup, execution, and management. SGX2 adds instructions such as `EAUG`, `EACCEPT`, and `EMODT`... for adding a page, accepting changes to a page, and modifying TCS structure (more on using these later).

Table 1. SGX1 supervisor and user mode instructions

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EXIT]	Exit an enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDGRD]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Reenter an enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		

Intel SGX enclave operation uses many data structures, including:

- Intel SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State State Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)
- Paging Crypto MetaData (PCMD)

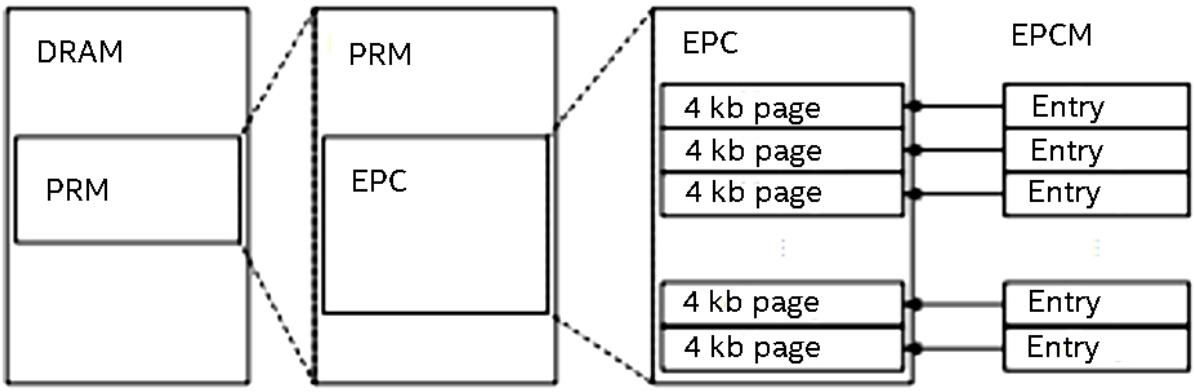
The structures such as SECS are unique per enclave and contain metadata that's immutable once it's instantiated and can only be accessed by the processor. TCS structure indicates the execution point into an enclave.

Enclave Page Cache

The enclave page cache (EPC) is secure storage used by the processor to store enclave pages. These 4KB pages can be marked either valid (if they belong to an enclave instance) or invalid. The metadata for the EPC pages is held in an internal microarchitecture structure called an enclave page cache map (EPCM).

The EPC is typically configured at BIOS/boot and the contents are encrypted (if allocated in DRAM). The pages are decrypted when they're inside the physical processor core. The keys are generated at boot and stored within the processor.

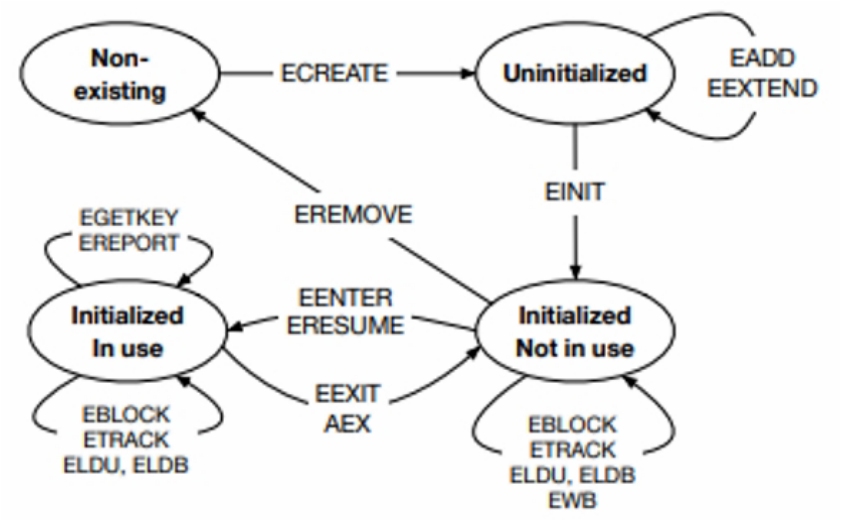
The Intel SGX instruction set has support for managing content on these EPC pages. The size of the EPC is implementation-specific (in some processors, it's 128MB maximum). The EPC is allocated in the processor reserved memory (PRM). **Figure 3** shows this hierarchy.



3 **Enclave data is stored in EPC, which is a subset of the PRM[3]**

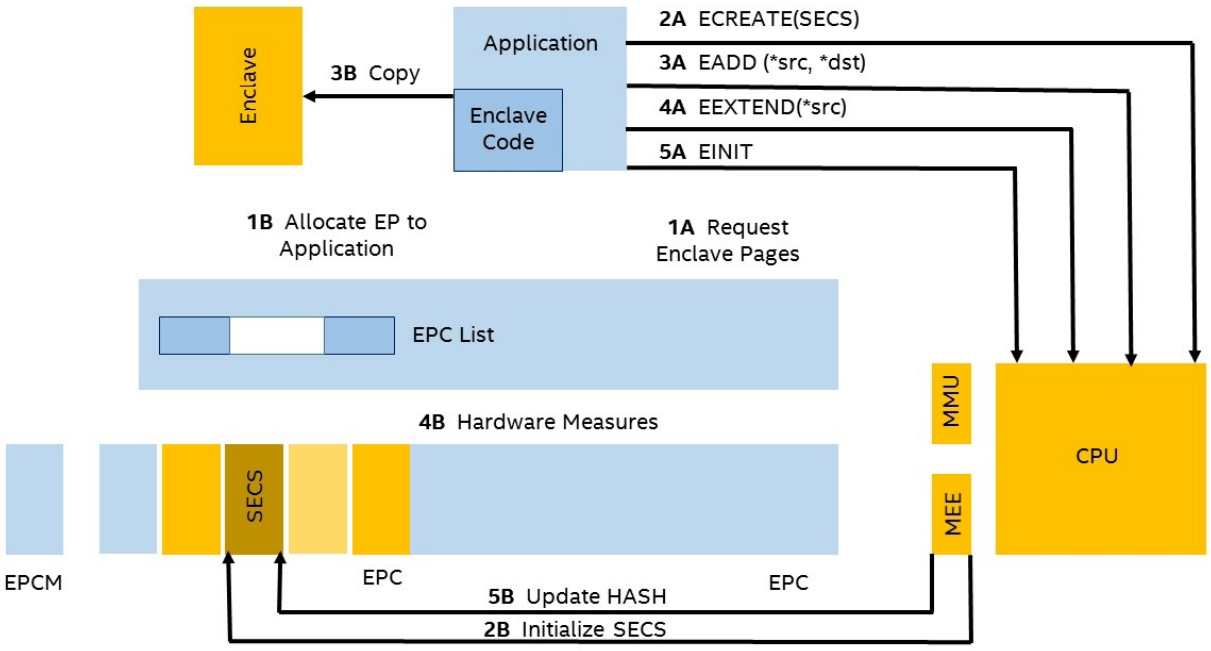
Enclave Operation

Figure 4 shows an enclave's lifecycle.[3] An enclave is born when the system software issues an `ECREATE` instruction, which allocates a free `EPC` page into the `SECS` for this new enclave. The `ECREATE` initializes the `SECS` structure with the required `BASE` address, `SIZE`, and other parameters and marks the enclave as uninitialized. Using the `EADD` instruction, the system software loads code and data into the enclave. Attempting to `EADD` a page on an initialized enclave would result in a fault.



4 **Intel SGX enclave lifecycle management instructions and state transition diagram[3]**

After loading the code and data pages, the enclave must be initialized using the `EINIT` instruction. After a successful initialization, the enclave can be invoked from application software. Using the `EREMOVE` instruction, an enclave can be destroyed and the corresponding EPC pages released. **Figure 5** shows the enclave creation steps and the corresponding state updates.[4] (For details about Intel SGX, check the references section.)



5 **Enclave creation steps[4]**

To establish trust, an enclave must do three activities:

1. **Measurement** to establish identity of the enclave
2. **Attestation** to demonstrate authenticity
3. **Sealing** to save it for later

An enclave measurement is accomplished by having it signed by the enclave author. Specifically, a 256-bit hash for the code and data, the author's public key, security version number, and the product ID of the enclave are all stored and compared. (You can find details on attestation and sealing in the references.)

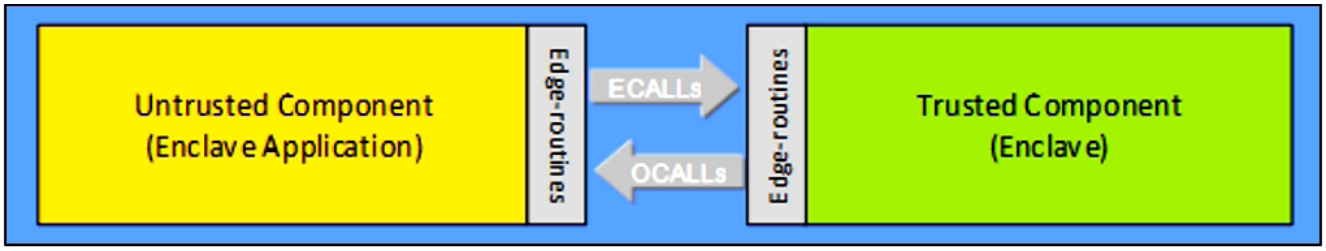
You can expect some performance impact with Intel SGX. How much depends on:

- **The number of enclave transitions** (recommended to be minimum)
- **The enclave memory access** (encryption/decryption)
- **EPC size** (paging issues)

Programming Intel SGX SDK

The first step in designing an Intel SGX-enabled application is to identify the assets (data, code) it needs to protect and place them in a separate trusted library (the enclave). This is no different from any regular application code except that the enclave code is loaded in a special way so that any untrusted application, system software, VMM, or OS cannot directly read data or change code within the enclave.

The first generation of the Intel SGX architecture requires all the functionality inside an enclave to be statically linked at build time. An enclave must expose an API for applications to call in (`ECALL`) and the services it needs from the untrusted domain (`OCALL`). **Figure 6** shows `ECALL`/`OCALL`. The `ECALL` and `OCALL` define the enclave boundary interface. To minimize the attack surface, the number of `ECALL` interfaces exposed should be limited.



6 **SGX application design considerations**

The elements of an Intel SGX runtime system would include an Untrusted Run-Time System (uRTS) and a Trusted Run-Time System (tRTS). The code within an enclave would constitute tRTS (receives ECALLs and makes OCALLs) and all the code outside of an enclave would be uRTS (makes ECALLs and receives OCALLs). The Intel SGX SDK provides APIs, libraries, and sample code to help developers write security applications using the Intel SGX technology. Using the Intel SGX SDK, you could create an Intel SGX enclave project with the necessary trusted/untrusted enclave C++ and EDL files.

An EDL (enclave definition language) file is used to define the enclave interface. These files would have an `.edl` file extension. The EDL files would define the interfaces and data types the enclave will support. There are two parts to the EDL file:

1. **The trusted section** with ECALLs
2. **The untrusted section** with OCALLs

At least one ECALL is needed to enter an enclave. An OCALL is optional. **Figure 7** shows the syntax of an EDL file. An EDL (library) and its functions can be selectively imported from other projects.

```
enclave{
        trusted{
          public void ecall_array_user_check([user+check] int arr[4]);
        };
        untrusted{
        };
};
```

7 An EDL file syntax/definition

Any Intel SGX application would first have to check the status of the device to see if Intel SGX is enabled using the `sgx_enable_device` API in the Platform Software (PSW). The next step would be to create an enclave using the `sgx_create_enclave` function (this is uRTS code). This call would load the enclave into the enclave memory. The `sgx_sign.exe` tool (in Intel SGX SDK) should be used to first sign the enclave that is being built as a DLL (Windows* dynamic link library) or Linux* shared object. Once the enclave is loaded, it's initialized by the tRTS after authenticating the enclave attributes. The uRTS and tRTS are part of the PSW and an Intel SGX application would have at least one project for each. The Intel SGX SDK has these tools to help build Intel SGX projects:

- Edger8r for generating trusted/untrusted interfaces
- Signing tool
- Debugger
- CPUSVN configuration tool
- Enclave Memory Measurement tool to measure usage of protected memory

Given the base EDL definitions/files, the Edger8r generates the trusted and untrusted bridges between the application and the enclave (with `_u.h/.c` and `_t.h/.c` file extensions). Data being sent to and received from an enclave is marked with attributes `[in]`, `[out]` or `[in, out]`. **Figures 8** through **10** show the enclave creation and execution steps for sample enclave code. A complex SGX application with more functions/interface definitions could be built along the same lines.[5]

```
enclave{
       from "sgx_tstdc.dl" import *;

trusted {

       /* define ECALLs here. */
       public void foo([out, size=len] char* buf, size_t len);
```

8 **Enclave1.edl sample code**

```
#include "Enclave1_t.h"

#include "sgx_trts.h"
#include <string.h>
void foo(char *buf, size_t len)
{
       const char *secret = "Hello Enclave!" ;
       if (len > strlen(secret))
          memcpy (buf, secret, strlen(secret) + 1);
```

9 **Enclave1.cpp sample code**

```
#include "sgx_urts.h"
#include "Enclave1_u.h"
#define ENCLAVE_FILE _T("enclave1.signed.dll")

Using namespace std;
Int main()
{

        sgx_enclave_id_t eid;
        sgx_status_t_ ret = SGX_SUCCESS;
        sgx_launch_token_t token = {0};
        int updated = 0;
        char buffer [MAX_BUF_LEN] = "Hello World!";
        // Create the Enclave with above launch token.
        ret = sgx_create_enclave(ENCLAVE_FILE, SGX_DEBUG_FLAG,
                &token, &updated,
                &eid, NULL);
        if (ret ! = SGX_SUCCESS) {
                printf{"App: error %#x, failed to create enclave. \n",
                        ret);
                return -1;
        }
        // A bunch of Enclave calls (ECALL) will happen here.
        foo(eid, buffer, MAX_BUF_LEN);
        printf("%s", buffer);
        //Destroy the enclave when all Enclave calls finish.
        if (SGX_SUCCESS != sgx_destroy_enclave(eid))
                return -1;
```

10 **The `Enclave1.dll` (and `Enclave1.signed.dll`) have "foo" (enclave) function defined and exposed for an untrusted console application (main) to execute.**

Increasing Application Security

Intel SGX provides a set of instructions that help increase the security of an application's code and data. This is accomplished by partitioning an application into enclaves (trusted regions with memory protection). The Intel SGX SDK provides APIs, libraries, and tools to help developers leverage the processor SGX features.

References

1. **Intel® Secure Guard Extensions SDK**
2. **Intel® SGX Programming Reference**
3. **Intel SGX Explained**
4. **Overview of Intel SGX - Part 1, SGX Internals**
5. **Intel® Software Guard Extensions Developer Guide**

VERIZON MAXIMIZES CUSTOMER SATISFACTION THROUGH BETTER PERFORMANCE

Optimizing Application Performance with Powerful Profiling

Guest Editorial by Dennis O'Connell, Senior Director of Performance Engineering, Verizon

Verizon is a global leader in delivering innovative communications and technology solutions that improve the way its customers live, work, and play. The Verizon Performance Engineering Group (PEG) is responsible for determining the configuration of servers and when to adopt new technology in the company's world-class datacenters. They're a leader in implementing next-gen technology—and also in helping Intel by providing invaluable feedback and direction to help applications to reach peak performance on the servers on which they're deployed.

Occasionally, PEG deals with code built on an older architecture that fails to capitalize on the latest technology breakthroughs. Profiling can be a powerful tool to quickly determine if there's potential to optimize hardware resources. The challenge is choosing among numerous profilers and performance analysis tools. An incorrect tool can waste time and offer no benefit to help interpret and publish the data gathered by profiling.

Working with experts both inside and outside the company, the PEG team has been able to identify the right profiling tools—which helps to improve the performance of key applications.

Solving Performance Problems

PEG is a team of both hardware and software engineers focused on solving performance problems for the infrastructure of the company's various business groups and subsidiaries. The goal is to make sure core workloads deliver optimum performance to produce the best user experiences for customers.

Business groups come to the PEG team when they have a problem with one of their workloads running on one of its servers. The PEG team will analyze the issue and provide the best solution for them within 24 hours.

"Our business units have excellent developers on their teams," explained PEG Principal Performance Engineer Mourad Bouache. "Occasionally, they'll come to us with latency issues on a core workload on a new server deployment. While they should be getting X performance, they're getting Y performance instead. We can quickly run an analysis and find where their code has issues slowing performance (e.g., NUMA memory contention). With small fixes in their code, they can get to X performance."

Pinpointing the Issues

To push the performance of C++, Java*, and Node.js* applications to the max, the team developed a new methodology for performance analysis based on **Intel® VTune™ Amplifier**, a software tool from Intel that's available standalone and as part of the **Intel® Parallel Studio XE** and **Intel® System Studio** tool suites.

Ensuring the best possible performance of systems for our users is a top priority for us. Intel VTune Amplifier helps us do that with effective workload management. It gives us abstracted information with ability to dive deeply with details such as hotspots, cache miss ratios, amount of concurrency, and lock contention mapped to function, source code line, and assembly instruction. By identifying issues that were otherwise overlooked, it allowed us to improve the performance of some of our crucial and revenue-impacting applications. Our team helps save the company tens of millions of dollars by using features like the Platform Profiler to manage performance issues on our servers and get useful insights into how we can achieve the highest level of performance from our hardware. We understand that we'll minimize our total cost of ownership not only with our very talented PEG team, but also by using world-class performance analysis tools like Intel VTune Amplifier.

Collaborating for the Future

The PEG team plans to continue collaborating with Intel to help ensure the tools meet our evolving needs—a collaboration that's valuable for both companies. "Intel looks to PEG as experts on improving application performance and great collaborators in making software tools that are as effective as possible," explained Sanjiv Shah, Vice President of Intel Architecture, Graphics, and Software. "They provide valuable feedback and help us define future enhancements to give both our customers and theirs the best possible performance."

Learn More

- **Intel® VTune™ Amplifier**

BLOG HIGHLIGHTS

Delivering "One Intel Developer Experience" Online

SCOTT HAY, INTEL CORPORATION

As Intel reaches out to the global software developer community, we aspire to deliver value and opportunity through our world-class One Developer Experience (ODX). From an online perspective, ODX means a single sign-on to our resources; a single taxonomy across our programs; easy access to our content, tools, and trusted support; and a true sense of excitement about what you can build, test, and deliver with the combination of Intel® hardware and software and the valuable resources and experiences gained from Intel's developer programs.

Read more >

COMPOSABLE THREADING IS COMING TO JULIA*

Flexible Parallelism in a Productivity Language

Editorial by Henry A Gabb, Senior Principal Engineer, Intel Corporation

The **July 2017** issue of *The Parallel Universe* ran an article on **Julia*: A High-Level Language for Supercomputing**. My key takeaways from the article were that Julia has built-in primitives for multithreading and distributed computing and is capable of extreme parallelism (scaling to thousands of cores). Despite this, I still considered Julia something of a curiosity. As I noted in my editorial in that issue, Python* was my productivity language of choice:

"I recoded some of my time-consuming data wrangling applications from Python to Julia, maintaining a line-for-line translation as much as possible. The performance gains were startling, especially because these were not numerically-intensive applications, where Julia is known to shine. They were string manipulation applications to prepare data sets for text mining. I'm not ready to forsake Python and its vast ecosystem just yet, but Julia definitely has my attention."

That was over two years ago, and Julia still has my attention because its ecosystem is growing rapidly and it has the potential to solve the "two-language problem" in data science, in which a high-level language like Python is used for data manipulation but low-level languages like Fortran or C/C++ are used when performance is required.

As much as I like Python, I'm not impressed with its ability to exploit parallelism, which brings me to the point of this brief commentary. Julia Computing **announced** last month that composable, general task parallelism is coming to Julia. (It's available now for testing in the **v1.3.0-alpha release**.) This is in addition to its already impressive parallel capabilities.

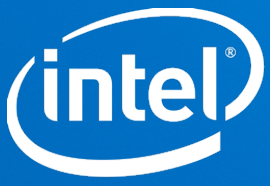
Julia's task parallelism is similar in spirit to that of **Threading Building Blocks**. The programmer spawns tasks freely and lets the scheduler sort out when and where they run. The programmer doesn't have to worry about available processors or threads because the scheduler makes sure that the system isn't oversubscribed. As the authors note, "The model is nestable and composable: you can start parallel tasks that call library functions that themselves start parallel tasks, and everything works." They illustrate this principle with a straightforward merge sort implemented with task parallelism:

```
import Base.Threads.@spawn

# sort the elements of 'v' in place, from indices 'lo' to 'hi' inclusive
function psort!(v, lo::Int=1, hi::Int=length(v))
   # some condition checking code (e.g., v is empty) deleted for brevity
   mid = (lo+hi)>>>1                  # find the midpoint
   half = @spawn psort!(v, lo, mid)   # task to sort the lower half; will run
   psort!(v, mid+1, hi)               # in parallel with the current call sorting
                                      # the upper half
   wait(half)                         # wait for the lower half to finish
   temp = v[lo:mid]                   # workspace for merging
   i, k, j = 1, lo, mid+1             # merge the two sorted sub-arrays
   @inbounds while k < j <= hi
      if v[j] < temp[i]
         v[k] = v[j]
         j += 1
      else
         v[k] = temp[i]
         i += 1
      end
      k += 1
   end
   @inbounds while k < j
      v[k] = temp[i]
      k += 1
      i += 1
   end
   return v
end
```

As you can see, Julia's syntax is similar to other productivity languages, so the learning curve is low. The `psort` function above is a standard merge sort with recursive spawning of threads (via the `@spawn` construct). Depending on the length of `v`, there's the potential to create many threads, but that's the Julia scheduler's problem. The programmer just specifies where to spawn threads and where to wait for them to finish.

I've only scratched the surface here, so have a look at their **original blog post** for more information about the new threading constructs and "under the hood" implementation details.



Software

THE PARALLEL UNIVERSE

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Configuration: Refer to Detailed Workload Configuration Slides in this presentation. Performance results are based on testing as of March 11th and March 25th 2019 and may not reflect all publicly available security updates. See configuration disclosures for details. No product can be absolutely secure. *Other names and brands may be claimed as property of others.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

* Other names and brands may be claimed as the property of others.

Printed in USA 1019/SS Please Recycle