Tuning Autonomous Driving Using Intel® System Studio

Julia*: A High-Level Language for Supercomputing

Vectorization Becomes Important—Again
# CONTENTS

<table>
<thead>
<tr>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Letter from the Editor</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Old and New</strong></td>
<td></td>
</tr>
<tr>
<td>by Henry A. Gabb, Senior Principal Engineer at Intel Corporation</td>
<td></td>
</tr>
<tr>
<td><strong>Tuning Autonomous Driving Using Intel® System Studio</strong></td>
<td>5</td>
</tr>
<tr>
<td>Intel® GO™ SDK Offers Automotive Solution Developers an Integrated Solutions Environment</td>
<td></td>
</tr>
<tr>
<td><strong>OpenMP® Is Turning 20!</strong></td>
<td>19</td>
</tr>
<tr>
<td>Making Parallel Programming Accessible to C/C++ and Fortran Programmers</td>
<td></td>
</tr>
<tr>
<td><em><em>Julia</em>: A High-Level Language for Supercomputing</em>*</td>
<td>23</td>
</tr>
<tr>
<td>The Julia Project Continues to Break New Boundaries in Scientific Computing</td>
<td></td>
</tr>
<tr>
<td><strong>Vectorizeration Becomes Important—Again</strong></td>
<td>31</td>
</tr>
<tr>
<td>Open Source Code WARP3D Exemplifies Renewed Interest in Vectorization</td>
<td></td>
</tr>
<tr>
<td><strong>Building Fast Data Compression Code for Cloud and Edge Applications</strong></td>
<td>40</td>
</tr>
<tr>
<td>How to Optimize Your Compression with Intel® Integrated Performance Primitives (Intel® IPP)</td>
<td></td>
</tr>
<tr>
<td><em><em>MySQL</em> Optimization with Intel® C++ Compiler</em>*</td>
<td>48</td>
</tr>
<tr>
<td>How LeCloud Leverages MySQL* to Deliver Peak Service</td>
<td></td>
</tr>
<tr>
<td><em><em>Accelerating Linear Regression in R</em> with Intel® DAAL</em>*</td>
<td>58</td>
</tr>
<tr>
<td>Make Better Predictions with This Highly Optimized Open Source Package</td>
<td></td>
</tr>
</tbody>
</table>

For more complete information about compiler optimizations, see our Optimization Notice.
Old and New

Back in 1993, the institute where I was doing my postdoctoral research got access to a Cray C90* supercomputer. Competition for time on this system was so fierce that we were told—in no uncertain terms—that if our programs didn’t take advantage of the architecture, they should run elsewhere. The C90 was a vector processor, so we had to vectorize our code. Those of us who took the time to read the compiler reports and make the necessary code modifications saw striking performance gains. Though vectorization would eventually take a backseat to parallelization in the multicore era, this optimization technique remains important. In Vectorization Becomes Important—Again, Robert H. Dodds Jr. (professor emeritus at the University of Illinois) shows how to vectorize a real application using modern programming tools.

We continue our celebration of OpenMP*’s 20th birthday with a guest editorial from Bronis R. de Supinski, chief technology officer of Livermore Computing and the current chair of the OpenMP Language Committee. Bronis gives his take on the conception and evolution of OpenMP as well as its future direction in OpenMP Is Turning 20! Though 20 years old, OpenMP continues to evolve with modern architectures.

I used to be a Fortran zealot, before becoming a Perl* zealot, and now a Python* zealot. Recently, I had occasion to experiment with a new productivity language called Julia*. 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. Take a look at Julia*: A High-Level Language for Supercomputing for an overview of the language and its features.
This issue's feature article, **Tuning Autonomous Driving Using Intel® System Studio**, illustrates how the tools in Intel System Studio give embedded systems and connected device developers an integrated development environment to build, debug, and tune performance and power usage. Continuing the theme of tuning edge applications, **Building Fast Data Compression Code for Cloud and Edge Applications** shows how to use the Intel® Integrated Performance Primitives to speed data compression.

As I mentioned in the last issue of *The Parallel Universe*, R* is not my favorite language. It is useful, however, and **Accelerating Linear Regression in R* with Intel® DAAL** shows how data analytics applications in R can take advantage of the Intel® Data Analytics Acceleration Library. Finally, in **MySQL* Optimization with Intel® C++ Compiler**, we round out this issue with a demonstration of how Interprocedural Optimization significantly improves the performance of another important application for data scientists, the MySQL database.

**Coming Attractions**

Future issues of *The Parallel Universe* will contain articles on a wide range of topics, including persistent memory, IoT development, the Intel® AVX instruction set, and much more. Stay tuned!

**Henry A. Gabb**
July 2017
The Internet of Things is a collection of smart devices connected to the cloud. “Things” can be as small and simple as a connected watch or a smartphone, or they can be as large and complex as a car. In fact, cars are rapidly becoming some of the world’s most intelligent connected devices, using sensor technology and powerful processors to sense and continuously respond to their surroundings. Powering these cars requires a complex set of technologies:

- **Sensors** that pick up LIDAR, sonar, radar, and optical signals
- **A sensor fusion hub** that gathers millions of data points
- **A microprocessor** that processes the data
- **Machine learning algorithms** that require an enormous amount of computing power to make the data intelligent and useful
Successfully realizing the enormous opportunities of these automotive innovations has the potential to not only change driving but also to transform society.

**Intel® GO™ Automotive Software Development Kit (SDK)**

From car to cloud—and the connectivity in between—there is a need for automated driving solutions that include high-performance platforms, software development tools, and robust technologies for the data center. With Intel GO automotive driving solutions, Intel brings its deep expertise in computing, connectivity, and the cloud to the automotive industry.

Autonomous driving on a global scale takes more than high-performance sensing and computing in the vehicle. It requires an extensive infrastructure of data services and connectivity. This data will be shared with all autonomous vehicles to continuously improve their ability to accurately sense and safely respond to surroundings. To communicate with the data center, infrastructure on the road, and other cars, autonomous vehicles will need high-bandwidth, reliable two-way communication along with extensive data center services to receive, label, process, store, and transmit huge quantities of data every second. The software stack within autonomous driving systems must be able to efficiently handle demanding real-time processing requirements while minimizing power consumption.

The Intel GO automotive SDK helps developers and system designers maximize hardware capabilities with a variety of tools:

- **Computer vision, deep learning, and OpenCL™ toolkits** to rapidly develop the necessary middleware and algorithms for perception, fusion, and decision-making
- **Sensor data labeling tool** for the creation of "ground truth" for deep learning training and environment modeling
- **Autonomous driving-targeted** performance libraries, leading compilers, performance and power analyzers, and debuggers to enable full stack optimization and rapid development in a functional safety compliance workflow
- **Sample reference applications**, such as lane change detection and object avoidance, to shorten the learning curve for developers
**Intel® System Studio**

Intel also provides software development tools that help accelerate time to market for automated driving solutions. **Intel System Studio** provides developers with a variety of tools including compilers, performance libraries, power and performance analyzers, and debuggers that maximize hardware capabilities while speeding the pace of development. It is a comprehensive and integrated tool suite that provides developers with advanced system tools and technologies to help accelerate the delivery of the next-generation, power-efficient, high-performance, and reliable embedded and mobile devices. This includes tools to:

- **Build** and optimize your code
- **Debug and trace** your code to isolate and resolve defects
- **Analyze** your code for power, performance, and correctness

**Build and Optimize Your Code**

- **Intel® C++ Compiler**: A high-performance, optimized C and C++ cross-compiler that can offload compute-intensive code to Intel® HD Graphics.
- **Intel® Math Kernel Library (Intel® MKL)**: A set of highly optimized linear algebra, fast Fourier transform (FFT), vector math, and statistics functions.
- **Intel® Threading Building Blocks (Intel® TBB)**: C++ parallel computing templates to boost embedded system performance.
- **Intel® Integrated Performance Primitives (Intel® IPP)**: A software library that provides a broad range of highly optimized functionality including general signal and image processing, computer vision, data compression, cryptography, and string manipulation.

**Debug and Trace Your Code to Isolate and Resolve Defects**

- **Intel® System Debugger**: Includes a System Debug feature that provides source-level debugging of OS kernel software, drivers, and firmware plus a System Trace feature that provides an Eclipse* plug-in, which adds the capability to access the Intel® Trace Hub providing advanced SoC-wide instruction and data events tracing through its trace viewer.
- **GNU* Project Debugger**: This Intel-enhanced GDB is for debugging applications natively and remotely on Intel® architecture-based systems.

**Analyze Your Code for Power, Performance, and Correctness**

- **Intel® VTune™ Amplifier**: This software performance analysis tool is for users developing serial and multithreaded applications.
- **Intel® Energy Profiler**: A platform-wide energy consumption analyzer of power-related data collected on a target platform using the SoC Watch tool.
- **Intel® Performance Snapshot**: Provides a quick, simple view into performance optimization opportunities.
- **Intel® Inspector**: A dynamic memory and threading error-checking tool for users developing serial and multithreaded applications on embedded platforms.
- **Intel® Graphics Performance Analyzers**: Real-time, system-level performance analyzers to optimize CPU/GPU workloads.
YOUR NEW SECRET WEAPON

EXPERIENCE UNMATCHED PERFORMANCE WITH INTEL® SYSTEM STUDIO.

Factory automation has never been so intuitive. Shorten validation cycles and maximize system-wide efficiency. It starts today.

Automate with precision ›

For more complete information about compiler optimizations, see our Optimization Notice.

Intel and the Intel logo 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
Optimizing Performance

Advanced Hotspot Analysis

Matrix multiplication is a commonly used operation in autonomous driving. Intel System Studio tools, mainly the performance analyzers and libraries, can help maximize performance. Consider this example of a very naïve implementation of matrix multiplication using two nested for-loops:

```c
void multiply0(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
              TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

    // Basic serial implementation
    for(i=0; i<msize; i++) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

Advanced hotspot analysis is a fast and easy way to identify performance-critical code sections (hotspots). The periodic instruction pointer sampling performed by Intel VTune Amplifier identifies code locations where an application spends more time. A function may consume much time either because its code is slow or because the function is frequently called. But any improvements in the speed of such functions should have a big impact on overall application performance.

Running an advanced hotspot analysis on the previous matrix multiplication code using Intel VTune Amplifier shows a total elapsed time of 22.9 seconds (Figure 1). Of that time, the CPU was actively executing for 22.6 seconds. The CPI rate (i.e., cycles per instruction) of 1.142 is flagged as a problem. Modern superscalar processors can issue four instructions per cycle, suggesting an ideal CPI of 0.25, but various effects in the pipeline—like long latency memory instructions, branch mispredictions, or instruction starvation in the front end—tend to increase the observed CPI. A CPI of one or less is considered good but different application domains will have different expected values. In our case, we can further analyze the application to see if the CPI can be lowered. Intel VTune Amplifier’s advanced hotspot analysis also indicates the top five hotspot functions to consider for optimization.
Elapsed time and top hotspots before optimization

**CPU Utilization**

As shown in Figure 2, analysis of the original code indicates that only one of the 88 logical CPUs is being used. This means there is significant room for performance improvement if we can parallelize this sample code.

CPU usage
Parallelizing the sample code as shown below gives an immediate 12x speedup (Figure 3). Also, the CPI has gone below 1, which is also a significant improvement.

```c
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
                TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;
    // Basic parallel implementation
    #pragma omp parallel for
    for(i=0; i<msize; i++) {
        for(j=0; j<msize; j++) {
            for(k=0; k<msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

**Elapsed Time**: 1.752s

<table>
<thead>
<tr>
<th>CPU Time</th>
<th>26.121s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Retired</td>
<td>88,962,000,000</td>
</tr>
<tr>
<td>CPI Rate</td>
<td>0.943</td>
</tr>
<tr>
<td>CPU Frequency Ratio</td>
<td>1.135</td>
</tr>
<tr>
<td>Total Thread Count</td>
<td>18</td>
</tr>
<tr>
<td>Paused Time</td>
<td>0s</td>
</tr>
</tbody>
</table>

**CPU Usage Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU usage value.

3 Performance improvement from parallelization
General Exploration Analysis

Once you have used Basic Hotspots or Advanced Hotspots analysis to determine hotspots in your code, you can perform General Exploration analysis to understand how efficiently your code is passing through the core pipeline. During General Exploration analysis, Intel VTune Amplifier collects a complete list of events for analyzing a typical client application. It calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems. 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). General Exploration analysis performs this estimate and breaks up all pipeline slots into four categories:

1. Pipeline slots containing useful work that issued and retired (retired)
2. Pipeline slots containing useful work that issued and canceled (bad speculation)
3. Pipeline slots that could not be filled with useful work due to problems in the front end (front-end bound)
4. Pipeline slots that could not be filled with useful work due to a backup in the back end (back-end bound)

Figure 4 shows the results of running a general exploration analysis on the parallelized example code using Intel VTune Amplifier. Notice that 77.2 percent of pipeline slots are blocked by back-end issues. Drilling down into the source code shows where these back-end issues occur (Figure 5, 49.4 + 27.8 = 77.2 percent back-end bound). Memory issues and L3 latency are very high. The memory bound metric shows how memory subsystem issues affect performance. The L3 bound metric shows how often the CPU stalled on the L3 cache. Avoiding cache misses (L2 misses/L3 hits) improves latency and increases performance.
General exploration analysis results

```
46 // Naive implementation
47 for(i=tidx; i<size; i+=numt) {
48     for(j=0; j<size; j++) {
49         for(k=0; k<size; k++) {
50             c[i][j] = c[i][j] + a[i][k] * b[k][j];
51         }
52     }
53 }
```

Identifying issues
Memory Access Analysis

The Intel VTune Amplifier’s Memory Access analysis identifies memory-related issues, like NUMA (non-uniform memory access) problems and bandwidth-limited accesses, and attributes performance events to memory objects (data structures). This information is provided from instrumentation of memory allocations/deallocations and getting static/global variables from symbol information.

By selecting the grouping option of the Function/Memory Object/Allocation stack (Figure 6), you can identify the memory objects that are affecting performance. Out of the three objects listed in the multiply1 function, one has a very high latency of 82 cycles. Double-clicking on this object takes you to the source code, which indicates that array “b” has the highest latency. This is because array “b” is using a column-major order. Interchanging the nested loops changes the access to row-major order and reduces the latency, resulting in better performance (Figure 7).

6 Identifying memory objects affecting performance

7 General exploration analysis results
We can see that although the sample is still back-end-bound, it is no longer memory-bound. It is only core-bound. A shortage in hardware compute resources, or dependencies on the software’s instructions, both fall under core-bound. Hence, we can tell that the machine may have run out of out-of-order resources. Certain execution units are overloaded, or there may be dependencies in the program’s data or instruction flow that are limiting performance. In this case, vector capacity usage is low, which indicates floating-point scalar or vector instructions are using only partial vector capacity. This can be solved by vectorizing the code.

Another optimization option is to use Intel Math Kernel Library, which offers highly optimized and threaded implementations of many mathematical operations, including matrix multiplication. The \texttt{dgemm} routine multiplies two double-precision matrices:

```c
void multiply5(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
               TYPE c[][NUM], TYPE t[][NUM])
{
    double alpha = 1.0, beta = 0.0;

    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                NUM, NUM, NUM,
                alpha, (const double *)b,
                NUM, (const double *)a,
                NUM, beta, (double *)c, NUM);
}
```

Performance Analysis and Tuning for Image Resizing

Image resizing is commonly used in operation in the autonomous driving space. For example, we ran Intel VTune Amplifier’s advanced hotspots analysis on an open-source OpenCV* version of image resize (Figure 8). We can see that the elapsed time is 0.33 seconds and the top hotspot is the \texttt{cv:HResizeLinear} function, which consumes 0.19 seconds of the total CPU time.
Advanced hotspot analysis

Intel IPP offers developers highly optimized, production-ready building blocks for image processing, signal processing, and data processing (data compression/decompression and cryptography) applications. These building blocks are optimized using the Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2) instruction sets. Figure 9 shows the analysis results for the image resize that takes advantage of the Intel IPP. We can see that the elapsed time has gone down by a factor of two, and since currently only one core is used, there is an opportunity for further performance improvement using parallelism via the Intel Threading Building Blocks.
Checking code with Intel® VTune™ Amplifier
Conclusion

The Intel System Studio tools in the Intel GO SDK give automotive solution developers an integrated development environment with the ability to build, debug and trace, and tune the performance and power usage of their code. This helps both system and embedded developers meet some of their most daunting challenges:

- **Accelerate software development** to bring competitive automated driving cars to market faster
- **Quickly target and help resolve defects** in complex automated driving (AD), advanced driver assistance systems (ADAS), or software-defined cockpit (SDC) systems
- **Help speed performance** and reduce power consumption

This is all provided in one easy-to-use software package.

---

**BLOG HIGHLIGHTS**

**Performance of Classic Matrix Multiplication Algorithm on Intel® Xeon Phi™ Processor System**

**BY SERGEY KOSTROV (INTEL)**

Matrix multiplication (MM) of two matrices is one of the most fundamental operations in linear algebra. The algorithm for MM is very simple, it could be easily implemented in any programming language, and its performance significantly improves when different optimization techniques are applied.

Several versions of the classic matrix multiplication algorithm (CMMA) to compute a product of square dense matrices are evaluated in four test programs. Performance of these CMMAs is compared to a highly optimized ‘cblas_sgemm’ function of the Intel® Math Kernel Library (Intel® MKL). Tests are completed on a computer system with Intel® Xeon Phi™ processor 72105 running the Linux Red Hat* operating system in ‘All2All’ Cluster mode and for ‘Flat’, ‘Hybrid 50-50’, and ‘Cache’ MCDRAM modes.

---

**DOWNLOAD A FREE TRIAL OF INTEL® SYSTEM STUDIO**

For more complete information about compiler optimizations, see our Optimization Notice.
I suppose that 20 years makes OpenMP* a nearly full-grown adult. In any event, this birthday provides us with a good opportunity to review the origins and evolution of the OpenMP specification. As the chair of the OpenMP Language Committee (a role that I have filled for the last nine years, since shortly after the release of OpenMP 3.0), I am happy to have the opportunity to provide this retrospective and to provide a glimpse into the health of the organization that owns OpenMP and into our future plans to keep it relevant. I hope you will agree that OpenMP remains true to its roots while evolving sufficiently to have reasonable prospects for another 20 years.

OpenMP arose in an era in which many compiler implementers supported unstandardized directives to guide loop-based, shared-memory parallelization. While these directives were often effective, they failed to meet a critical user requirement: portability. Not only did the syntax (i.e., spelling) of those implementation-specific directives vary but they also often exhibited subtle differences in their semantics. Recognizing this deficiency, Mary Zosel from Lawrence Livermore National Laboratory worked closely with the implementers to reach agreement on common
syntax and semantics that all would provide in OpenMP 1.0. In addition, they created the OpenMP organization to own and to maintain that specification.

Many view OpenMP's roots narrowly as exactly the loop-based, shared-memory directives that were standardized in that initial specification. I prefer to look at them more broadly as a commitment to support portable directive-based parallelization and optimization, possibly limited to within a process, through an organization that combines compiler implementers with user organizations who require their systems to deliver the highest possible portable performance. Ultimately, I view OpenMP as a mechanism for programmers to express key performance features of their applications that compilers would find difficult or even impossible to derive through (static) analysis. Organizationally, OpenMP is vibrant, with membership that includes representatives of all major compiler implementations (at least in the space relevant to my user organization) and an active and growing set of user organizations.

Technically, the OpenMP specification met its original goal of unifying the available loop-based parallelization directives. It continues to provide simple, well-defined constructs for that purpose. Further, OpenMP has fostered ongoing improvements to those directives, such as the ability to collapse loops and to control the placement of threads that execute the parallelized code. Achieving consistently strong performance for these constructs across shared-memory architectures and a complete range of compilers, OpenMP provides the portability that motivated its creation.

OpenMP's evolution has led it to adopt several additional forms of parallelism. I am often annoyed to hear people in our community say that we need a standardized mechanism for task-based parallelism. OpenMP has provided exactly that for the last nine years! In 2008, the OpenMP 3.0 specification was adopted with a complete task-based model. While I acknowledge that OpenMP tasking implementations could still be improved, we face a chicken-and-the-egg problem. I often hear users state that they will use OpenMP tasking when implementations consistently deliver strong performance with the model. However, implementers also frequently state that they will optimize their tasking implementation once they see sufficient adoption of the model. Another issue for the OpenMP tasking model derives from one of OpenMP's strengths—it provides consistent syntax and semantics for a range of parallelization strategies that can all be used in a single application. Supporting that generality necessarily implies inherent overheads. However, model refinements can alleviate some of that overhead. For example, OpenMP 3.1 added the final clause (and its related concepts) to facilitate specifying a minimum task granularity, which otherwise requires complex compiler analysis or much more complex program structure. As we work toward future OpenMP specifications, we are continuing to identify ways to reduce the overhead of OpenMP tasking.
OpenMP added support for additional forms of parallelism with the release of OpenMP 4.0. In particular, OpenMP now includes support for accelerator-based parallelization through its device constructs and for SIMD (or vector) parallelism. The support for the latter is particularly reminiscent of OpenMP's origins. Support for SIMD parallelism through implementation-specific directives had become widespread, most frequently in the form of `ivdep`. However, the supported clauses varied widely and, in some cases, the spelling of the directives was also different. More importantly, the semantics often varied in subtle ways that could lead directives to be correct with one compiler but not with another. The addition of SIMD constructs to OpenMP largely solves these problems, just as the creation of OpenMP 20 years ago solved them for loop-based directives. Of course, the initial set of SIMD directives is not perfect and we continue to refine them. For example, OpenMP 4.5 added the `simdlen` clause, which allows the user to specify the preferred SIMD vector length to use.

The OpenMP target construct specifies that computation in a structured block should be offloaded to a device. Additional constructs that were added in the 4.0 specification support efficient parallelization on devices such as GPUs. Importantly, all OpenMP constructs can be used within a target region, which means that the years of advances in directive-based parallelization are available for a range of devices. While one still must consider which forms of parallelism will best map to specific types of devices, as well as to the algorithms being parallelized, the orthogonality of OpenMP constructs greatly increases the portability of programs that need to target multiple architectures.

We are actively working on the OpenMP 5.0 specification, with its release planned for November 2018. We have already adopted several significant extensions. OpenMP TR4 documents the progress on OpenMP 5.0 through last November. We will also release a TR this November that will document our continued progress. The most significant addition in TR4 is probably OMPT, which extends OpenMP with a tool interface. I anticipate that OpenMP 5.0 will include OMPD, an additional tool interface that will facilitate debugging OpenMP applications. TR4 included many other extensions, perhaps most notably the addition of task reductions.

We expect that OpenMP 5.0 will provide several major extensions that were not included in TR4. Perhaps most notably, OpenMP will greatly enhance support for complex memory hierarchies. First, I expect it to include some form of the memory allocation mechanism that TR5 documents. This mechanism will provide an intuitive interface for programmers to portably indicate which memory should be used for particular variables or dynamic allocations in systems with multiple types of memory (e.g., DDR4 and HBM). Further, I expect that OpenMP will provide a powerful serialization and deserialization interface that will support deep copy of complex data structures. This interface will eventually enable data-dependent layout transformations and associated optimizations with device constructs.
As I have already mentioned, the OpenMP organization is healthy. We have added several members in the past few years, both implementers and user organizations. We are currently discussing changes to the OpenMP bylaws to support even wider participation. If you are interested in shaping the future of OpenMP, we are always happy to have more participation! Visit openmp.org for more information about OpenMP.

Bronis R. de Supinski is the chief technology officer of Livermore Computing, the supercomputing center at Lawrence Livermore National Laboratory. He is also the chair of the OpenMP Language Committee.
High-performance computing (HPC) has been at the forefront of scientific discovery. Scientists routinely run simulations on millions of cores in distributed environments. The key software stack involved has typically been a statically compiled language such as C/C++ or Fortran, in conjunction with OpenMP*/MPI*. This software stack has stood the test of time and is almost ubiquitous in the HPC world. The key reason for this is its efficiency in terms of the ability to use all available compute resources while limiting memory usage. This level of efficiency has always been beyond the scope of more “high-level” scripting languages such as Python*, Octave*, or R*. This is primarily because such languages are designed to be high-productivity environments that facilitate rapid prototyping—and are not designed to run efficiently on large compute clusters. However, there is no technical reason why this should be the case, and this represents a tooling problem for scientific computing.
Julia,* a new language for technical computing, is meant to address this problem. It reads like Python or Octave, but performs as well as C. It has built-in primitives for multithreading and distributed computing, allowing applications to scale to millions of cores.

The first natural questions that spring to mind are: Why can't the other high-level languages perform this well? What's unique about Julia that lets it achieve this level of efficiency while still allowing the user write readable code?

The answer is that Julia uses a combination of type inference and code specialization to generate tight machine code. To see how, let's use the Julia macro @code_native, which generates the assembly code generated by a function call. For example:

```
julia> @code_native 1 + 1
.section __TEXT,__text,regular,pure_instructions
Filename: int.jl
pushq %rbp
movq %rsp, %rbp
Source line: 32
leaq (%rdi,%rsi), %rax
popq %rbp
retq
Source line: 32
nopw (%rax,%rax)
```

This example shows the assembly code generated when Julia adds two integers. Notice that Julia picks the right assembly instruction for the operation, and there is virtually no overhead.

The next code snippet shows the addition of two double-precision floating-point numbers:

```
julia> @code_native 1. + 1.
.section __TEXT,__text,regular,pure_instructions
Filename: float.jl
pushq %rbp
movq %rsp, %rbp
Source line: 240
addsd %xmm1, %xmm0
popq %rbp
retq
Source line: 240
nopw (%rax,%rax)
```

Julia automatically picks a different instruction, addsd, since we are now adding two floating-point numbers. This is because two versions of the function "+" are generated depending on the type of the input, which was inferred by the compiler. This idea allows the user to write high-level code and then let Julia infer the types of the input and output, and subsequently compile a
specialized version of that function for those input types. This is what makes Julia fast (Figure 1).

1 Julia benchmark times relative to C (smaller is better). C performance = 1.0. Benchmark code can be found at the [GitHub repository](#).

Julia's unique features solve the **two-language problem** in data science and numerical computing. This has led to demand for Julia in a **variety of industries** from **aerospace** to **finance**.

This led to the formation of **Julia Computing**, which consults with industry and fosters adoption of Julia in the community. One of Julia Computing's flagship products is **JuliaPro***, a Julia distribution that includes an IDE, a debugger, and more than 100 tested packages. JuliaPro will soon ship with **Intel® Math Kernel Library (Intel® MKL)** for accelerated BLAS operations and optimizations for multicore and the latest Intel® processors.
Parallel Computing in Julia

Julia has several built-in primitives for parallel computing at every level: vectorization (SIMD), multithreading, and distributed computing.

Consider an embarrassingly parallel simulation as an example. An iterative calculation of π involves generating random numbers between 0 and 1 and eventually calculating the ratio of the number of points that “land” inside the unit circle as opposed to those that don’t. This gives the ratio of the areas of the unit square and the unit circle, which can then be used to calculate π. The following Julia code makes use of the `@parallel` construct. The reducer “+” sums the values of the final expression that was computed in parallel on all Julia processes:

```julia
addprocs(2)  # Add 2 Julia worker processes

function parallel_π(n)
    in_circle = @parallel (+) for i in 1:n  # <-- partition the work
        x = rand()
        y = rand()
        Int((x^2 + y^2) < 1.0)
    end
    return (in_circle/n) * 4.0
end

parallel_π(10000)
```

Julia offloads work to its worker processes that compute the desired results and send them back to the Julia master, where the reduction is performed. Arbitrary pieces of computation can be assigned to different worker processes through this one-sided communication model.

A more interesting use-case would be having different processes solve, for example, linear equations and then serialize their output and send it to the master process. Scalable machine learning is one such example which involves many independent backsolves. Figure 2 shows the performance of `RecSys.jl`, which processes millions of movie ratings to make predictions. It is based on an algorithm called Alternating Least Squares (ALS), which is a simple, iterative method for collaborative filtering. Since each solve is independent of every other, this code can be parallelized and scaled easily. Figure 2 is a performance chart that shows the scaling characteristics of the system.
In Figure 2, we look at the scaling performance of Julia in multithreaded mode (Julia MT), distributed mode (Julia Distr), and shared memory mode (Julia Shared). Shared memory mode allows independent Julia worker processes (as opposed to threads) to access a shared address space. An interesting aspect of this chart is the comparison between Julia's distributed capabilities and that of Apache Spark*, a popular framework for scalable analytics that is widely adopted in the industry. What's more interesting, though, are the consequences of this experiment: Julia can scale well to a higher number of nodes. Let's now move on to a real experiment in supercomputing.
Bringing It All Together (at Scale): Celeste*

The Celeste* Project is a collaboration of Julia Computing (Keno Fischer), Intel Labs (Kiran Pamnany), JuliaLabs@MIT (Andreas Noack, Jarrett Revels), Lawrence Berkeley National Labs (David Schlegel, Rollin Thomas, Prabhat), and University of California–Berkeley (Jeffrey Regier, Maximilian Lam, Steve Howard, Ryan Giordano, Jon McAuliffe). Celeste is a fully generative hierarchical model that uses statistical inference to mathematically locate and characterize light sources in the sky. This model allows astronomers to identify promising galaxies for spectrograph targeting, define galaxies for further exploration, and help understand dark energy, dark matter, and the geometry of the universe. The data set used for the experiment is the Sloan Digital Sky Survey (SDSS) (Figure 3), with more than five million images comprising 55 terabytes of data.

![A sample from the Sloan Digital Sky Survey (SDSS)](image)
Using Julia’s native parallel computing capabilities, researchers were able to scale their application to 8,192 Intel® Xeon® processor cores on the National Energy Research Scientific Computing Center (NERSC) Cori* supercomputer at LBNL. This resulted in parallel speedups of 225x in image analysis, processing more than 20,000 images (or 250 GB), an increase of three orders of magnitude compared to previous iterations.

Note that this was achieved with Julia’s native parallel capabilities, which allowed scaling of up to 256 nodes and four threads per node. This kind of scaling puts Julia firmly in the HPC realm, joining the elite league of languages such as C/C++ and Fortran.

Pradeep Dubey of Intel Labs summarized this nicely: “With Celeste, we are closer to bringing Julia into the conversation because we’ve demonstrated excellent efficiency using hybrid parallelism—not just processes, but threads as well—something that’s still impossible to do with Python or R.”

The next step in the project is scaling even further and processing the entire data set. This will involve making use of the latest Intel® Xeon Phi™ processors and coprocessors.

**Breaking New Boundaries**

The Julia project has come a long way since its inception in 2012, breaking new boundaries in scientific computing. Julia was conceived as a language that retained the productivity of Python while being as fast and scalable as C/C++. The Celeste Project shows that this approach allows Julia to be the first high-productivity language to scale well on clusters.

The Julia project continues to double its user base every year and to be increasingly adopted in a variety of industries. Such advances in tooling for computational science not only bode well for scientific research to address the challenges of the coming age, but will also result in rapid innovation cycles and turnaround times in the industry.
CODE THAT PERFORMS AND OUTPERFORMS

What could your code do with the speed of Intel® Parallel Studio XE?

Get ready for faster profiling, advanced compilers, and all the latest in threading and vectorization tools to propel your code further than ever. Outsmart performance bottlenecks and advance your code with Intel Parallel Studio XE.

Kick up your code's momentum ›
Once the mainstay of high-performance computing, vectorization has reemerged with new importance given the increasing width of SIMD (vector) registers on each processor core: four doubles with AVX2, eight with AXV512. Increased vector lengths create opportunities to boost performance in certain classes of software (e.g., large-scale finite element analysis to simulate the nonlinear behavior of mechanical systems). The open source research code, WARP3D, also on GitHub, focuses on such simulations to support the development of safety and life prediction methodologies for critical components found typically in energy production and related industries. The discipline emphasis (3-D nonlinear solids), open source, and manageable code size (1,400 routines) appeal to researchers in industry and academia.

WARP3D reflects the current hierarchy of parallel computing: MPI to access multiple nodes in a cluster, threads via OpenMP* on each node, and vectorization within each thread. Realistic simulations to predict complex behavior (e.g., the internal stresses created in welding a
large component) require many hours of parallel computation on current computers. The pervasive matrix operations to achieve solutions map very well onto this hierarchy of parallel computing. Increased use of vectorization at the core level provides a direct multiplying effect on performance across concurrently executing threads and MPI ranks.

This article provides a brief view into vectorization as implemented in WARP3D. Developed continuously over the last 20 years, the code adopts many of the evolving features in Fortran. We employ components of Intel® Parallel Studio XE (Intel® Fortran Compiler, Intel® Math Kernel Library, and Intel® MPI Library) to build executables included with the open source distribution for Linux*, Windows*, and MacOS*. WARP3D follows an implicit formulation of the nonlinear finite element method that necessitates solutions for very large, evolving sets of linear equations—the PARDISO* and CPARDISO* packages in Intel Math Kernel Library provide exceptional performance. Examples described here also illustrate the new Intel® Advisor 2017 with roofline analysis that enables more convenient and in-depth exploration of loop-level performance.

Figure 1 illustrates the decomposition of a 3-D finite element model for a flange into domains of elements (one domain/MPI rank) with elements in a domain assigned to blocks. Each discrete element in the model has an associated number of generally small permanent and temporary arrays (e.g., 6x6, 24x24, 60x60, 6x60), leading to millions of such arrays present during the solution of moderate- to large-sized models.

Figure 2A shows a conventional, multilevel array of structures (AoS) for storage of element data, with the various arrays for a single element likely contiguous in memory. While conceptually convenient, the comparatively small, element-level matrices limit inner-loop vector lengths and provide no mechanism to tune for cache size effects while processing the same array type across all elements.
The alternative, blocked data structure, as shown in Figure 2B for the element stiffness matrices $K_e$ and $B_e$ (as examples), defines a structure of arrays (SoA) scheme using dense 2-D and 3-D arrays with the leading dimension set by the number of elements assigned to the block, termed span in WARP3D. All the $K_e$ matrices for elements in a block, and separately the $B_e$ matrices, are thus contiguous in memory.

With a master list setting block ownership by domains, the following sequence illustrates the concept of combined MPI plus thread parallelism in WARP3D:

```c
$OMP PARALLEL DO PRIVATE(blk)
   do blk = 1, number_ele_blks
      if( blk_to_domain_map(blk) .ne. myrank ) cycle
      call process_a_blk(blk)
   end do
$OMP END PARALLEL DO
```

WARP3D employs a simple manager-worker design for MPI to synchronize tasks across ranks. In the above, the manager (on rank 0) ensures that all workers execute the OpenMP parallel loop to process the blocks they own. The `process_a_blk` routine dynamically allocates working versions of blocked arrays as needed in a derived type, declared local to the routine and thus local in each thread. The generally complex, multiple levels of computational routines to build/update element matrices use vector-style coding—meaning they are unaware of the OpenMP and MPI runtime environments. The data independence of element-based computations implied here represents a key feature of the finite element method. Global arrays are accessed read-only by the threads; the few block→gather operations on one vector specify `$OMP ATOMIC UPDATE` for synchronization. All threading operations in WARP3D are accomplished with only a dozen loops/directives of the type shown above.
The introduction of Cray® vector-based computers motivated the development of the blocking approach described above in finite element codes first created by the U.S. national laboratories. As these vector-based computers became extinct, the codes evolved to process element blocks in parallel using threads and then across MPI ranks, with blocking retained as a means to tune for effects of various cache architectures.

An example here illustrates essential features and performance of the conventional AoS and blocking SoA approaches. A sequence of matrix multiplications updates an array for each element in the model. For a single element, matrices $B_e$, $D_e$, and $K_e$ in this illustration have sizes 6x60, 6x6, 60x60 (all doubles and dense). Figure 2B shows the blocked storage scheme where the large arrays are allocatable components of a Fortran derived type. In models having different element types, and thus matrix sizes, WARP3D creates homogeneous blocks with all elements of the same type. Updates to $K_e$ have the form: $K_e \leftarrow K_e + \text{transpose}(B_e)D_eB_e$. This update occurs 300M to 1B times during various simulations of the flange ($D_eB_e$ also change for every update). In this example, $D_e$ and $K_e$ are symmetric, thus reducing computation to only lower-triangle terms.
Figure 3A shows essential code for the conventional AoS scheme. The computational outer loop cycles sequentially over all elements in the model with different versions of the called subroutine for symmetry, asymmetry, etc., while inner loops in the subroutine cycle over the much smaller dimensions of a single element matrix. The Fortran intrinsic `matmul` at line 19 performs the first matrix product, with loops over the comparatively small element arrays to compute final updates for only the lower-triangle terms of $K_e$. The compiler unrolls the $k=1,6$ inner loop at lines 23 to 25 and appears to replace matmul with equivalent source code which is then compiled—based on the extensive messages about loop interchanges in the annotated optimization report issued at line 19. [Editor's note: See “Vectorization Opportunities for Improved Performance with Intel® AVX-512” in The Parallel Universe issue 27 for advice on generating and interpreting compiler optimization reports.]

Code for the blocked SoA scheme in Figure 3B runs computational loops in this order:

1. Over all blocks (not shown)
2. Over dimensions of a single element matrix
3. The innermost loops over the number of elements (span) in the block

The first set of loops at lines 10 to 28 computes the product $\mathbf{db} = \mathbf{D}_e \mathbf{B}_e$ in a provided workspace for all elements in the block. A straightforward implementation would specify a four-level loop structure. Here, the two innermost loops have manual unrolling that essentially forces the compiler to vectorize the generally longer $i=1,\text{span}$ loop. Other arrangements explored for these loops increase runtimes; the compiler does not know the iteration count for each loop, which limits the effectiveness of loop reordering. The second set of loops at lines 30 to 42 performs the update $\mathbf{K}_e \leftarrow \mathbf{K}_e + \text{transpose} (\mathbf{B}_e) \mathbf{db}$. Here, $\text{utsz}=1820$ with index vector $\text{icp}$ to map lower-triangle terms for an element onto columns of the blocked $\mathbf{K}_e$. Again, the innermost loop has been manually unrolled for vectorization $i=1,\text{span}$.

Both sets of loops in this routine, lines 10 to 28 and 30 to 42, have only unit stride array references and long (span) lengths set up for vectorization, a characteristic feature throughout the WARP3D code. These are generally the most time-consuming loops for element-level processing in WARP3D; multiple redesigns of the loops and array organizations have occurred over the years with the evolution of compiler optimization and processor capabilities.
Figure 4 summarizes the relative execution performance for both approaches running on a single thread, with the AoS scheme assigned a unit value. Numbers in parentheses indicate the size for arrays in the compute routines. Relevant compiler options are: ifort (17.0.2) with `-O3 -xHOST -align array64byte -ftz`. The computer has dual Intel® Xeon® E5-2698 v3 processors (with AVX2) with no other user software running during the tests. This processor has cache sizes L1 = 32 KB/core, L2 = 256 KB/core, and L3 = 40 MB shared. Driver code and compute routines reside in separate files to prevent inlining effects. The runtimes include efforts to call the computation routines but not driver setup times. The driver program allocates thousands of array sets for both AoS and SoA testing. On each call, the driver passes a randomly selected set to the compute routine, in all cases processing 32 M elements. Multiple runs, each completing in two to four minutes, reveal a 1 to 2 percent variability in measured CPU.

Only a single result is needed for the AoS scheme; there are no adjustable parameters. The number of elements per block (span) varies in the SoA scheme with a compensating number of blocks to maintain the identical number of elements processed in each case. The SoA results show an expected strong effect of block size driven by the interaction of vector processing lengths and L1, L2, and L3 cache utilization. Runtimes reach a minimum of 37 percent of the AoS time for block sizes of 64 and 32 elements. At 16 elements/block, runtimes begin to increase. For all block sizes, the arrays fit within the L3 cache; the much smaller, single-element arrays of AoS almost fit within the L2 data cache.
Figure 5 shows output from the roofline analysis performed by Intel Advisor 2017 with results from the key cases manually superposed onto a single plot for comparison. Note the log scale on both axes. The roofline analysis with the graphical tools as implemented in Intel Advisor provides a convenient mechanism to screen the performance of key loops across large codes (note the hamburger menu in upper-right corner). Pointer movement over the symbols creates pop-ups with loop location in source code, GFLOPS, and floating-point operations per byte of L1 cache. Clicking on the symbol brings up source code for the loop annotated with additional performance information (e.g., if peel and remainder loops were executed for the loop).

During the inner loop of the $K_e$ update at lines 33 to 41, the best performance of 8.08 GFLOPS for block size (span) = 64 resides effectively on the roofline for the L2 cache but well below the peak rate for Fused-Multiply-Add (FMA) vector instructions, which are the kernel for this loop. The inner loop of the $D_eB_e$ computation at lines 11 to 27 gains a surprising 3.7x performance boost when the block size decreases from 1024→128 but no further gain for smaller blocks. The longest running loop at lines 33 to 41 gains a 2x performance increase for block size reductions 1024→64.
As expected from the timing results (Figure 4), GFLOPS and cache utilization for AoS shown by the blue filled and outlined squares are smaller than those for the blocked SoA. Replacement of code in the routine with the direct $ek = ek + \text{matmul}(\text{transpose}(b), \text{matmul}(d,b))$ increases runtime by 30 percent above the version in Figure 3A.

The best performance of 8.08 GFLOPS relative to the double-precision FMA peak of 51.4 GFLOPS shown on the roofline derives from the small number of floating point operations per memory access. This is a well-known characteristic of element-level computations inherent in the finite element method. Consider the key inner loop at lines 33 to 41, which has 12 double operations (6 + and 6 *) and 13 references to double-precision array terms. The compute intensity per byte is then, $12/(13*8) = 0.11$, the same value reported in the roofline analysis as FLOP/L1-byte. This value remains unchanged with block size. For reference, the figure also indicates the best performing routine in MKL BLAS as displayed by the roofline analysis during equation solving in WARP3D executions (single thread).

A few final points from the benchmarks:

- The roofline tab for Code Analytics lists: FMA as the only “trait” for the loop at lines 33 to 41, an 81 percent vectorization efficiency (3.24x gain over scalar), vector length = 4 double (AXV2). Further, the roofline tab for Memory Access Patterns confirms stride-1 access of all arrays for both inner loops in the routine.

- Memory alignment. The compile process uses the option `–align array64byte` needed for best performance on processors with AXV-512 instructions. Thirty-two-byte alignment suffices for the AVX2 capable processor used here. Nevertheless, with the separate compilation of the driver.f and routines.f files representative of build processes, the compiler is unaware of the actual alignment of the formal array arguments to the subroutine bdbtSoA. These adjustable arrays in the Fortran context guarantee to the compiler that array elements are contiguous in memory, but not necessarily with a specific alignment. The annotated source listing shows unaligned access to all arrays as expected.

Experiments with source code directives of the type `!DIR$ ASSUME_ALIGNED b(1,1,1):64, d(1,1,1):64, ...` or, separately, `!DIR$ VECTOR ALIGNED` before the two inner loops yield no measurable improvement in performance. The compiler listing indicates aligned access for all arrays.

The explanation may lie in the Intel Advisor messages that so-called peel and remainder loops, used to execute separately the initial loop iterations until access becomes aligned, are not executed here even without source code directives. At runtime, the alignment of arrays before entering the loop is easily determined to be 64 bytes in all cases here from the `–align array64byte` option in compiling driver.f.

Unit stride. A simple experiment demonstrates the critical significance of stride-1 array references on performance (at least on this processor). A switch in design/use of array $ek\_symm(span,utsz)$ to $ek\_symm(utsz,span)$ with $ek\_symm(i,j)$ changed to $ek\_symm(j,i)$ at line 34 reduces performance from 8.08 GFLOPS to 2.39 GFLOPS.
Summary

Vectorization offers potential speedups in codes with significant array-based computations—speedups that amplify the improved performance obtained through higher-level, parallel computations using threads and distributed execution on clusters. Key features for vectorization include tunable array sizes to reflect various processor cache and instruction capabilities and stride-1 accesses within inner loops.

The importance of vectorization to increase performance will continue to grow as hardware designers extend the number of vector registers to eight doubles (and hopefully more) on emerging processors to overcome plateauing clock rates and thread scalability.

Intel Advisor, especially the new roofline analysis capability, provides relevant performance information and recommendations implemented in a convenient GUI at a level suitable for code developers with differing levels of expertise.

Acknowledgments

The author gratefully acknowledges the many, key contributions of his former graduate students, postdocs, and other collaborators worldwide toward making WARP3D a useful code for researchers. See warp3d.net for a complete list and sponsors. These latest efforts toward improving code performance are in collaboration with Drs. Kevin Manalo and Karen Tomko, supported by the Intel® Parallel Computing Center at the Ohio Supercomputer Center.

References


About the author

Robert H. Dodds Jr., PhD, NAE, is the M.T. Geoffrey Chair (Emeritus) of Civil Engineering at the University of Illinois at Urbana–Champaign. He currently is a research professor in the Department of Civil Engineering, University of Tennessee, Knoxville.
Finding efficient ways to compress and decompress data is more important than ever. Compressed data takes up less space and requires less time and network bandwidth to transfer. In cloud service code, efficient compression cuts storage costs. In mobile and client applications, high-performance compression can improve communication efficiency—providing a better user experience.

However, compression and decompression consume processor resources. And, especially in data-intensive applications, this can negatively affect the overall system performance. So an optimized implementation of compression algorithms plays a crucial role in minimizing the system performance impact. **Intel® Integrated Performance Primitives (Intel® IPP)** is a library that...
contains highly optimized functions for various domains, including lossless data compression. In this article, we'll discuss these functions and the latest improvements. We will examine their performance and explain how these functions are optimized to achieve the best performance. We will also explain how applications can decide which compression algorithm to use based on workload characteristics.

**Intel® IPP Data Compression Functions**

The Intel IPP Data Compression Domain provides optimized implementations of the common data compression algorithms, including the BZIP2*, ZLIB*, LZO*, and a new LZ4* function, which will be available in the Intel IPP 2018 update 1 release. These implementations provide “drop-in” replacements for the original compression code. Moving the original data compression code to an Intel IPP-optimized code is easy.

These data compression algorithms are the basic functions in many applications, so a variety of applications can benefit from Intel IPP. A typical example is Intel IPP ZLIB compression. ZLIB is the fundamental compression method for various file archivers (e.g., gzip*, WinZip*, and PKZIP*), portable network graphics (PNG) libraries, network protocols, and some Java* compression classes. Figure 1 shows how these applications can adopt the optimized ZLIB in Intel IPP. Intel IPP provides fully compatible APIs. The application simply needs to relink with the Intel IPP ZLIB library. If the application uses ZLIB as a dynamic library, it can be easily switched to a new dynamic ZLIB library built with Intel IPP. In the latter case, relinking the application is not required.

In recent releases, the Intel IPP data compression domains extended several functionalities and increased compression performance:

- **Support for LZ4 data compression** will be included in Intel IPP 2018 beta update 1, available soon.
- **Increased LZO compression performance by 20 to 50 percent.** Added level 999 support in LZO decompression.
- **A new fastest ZLIB data compression mode**, which can improve performance by up to 2.3x compared to ZLIB level 1 performance.
- **Trained Huffman tables** for ZLIB compression let users improve compression ratio in the fastest mode.

---

For more complete information about compiler optimizations, see our Optimization Notice.
Intel IPP Data Compression Optimization

How does Intel IPP data compression perform the optimization and achieve top performance? Intel IPP uses several different methods to optimize data compression functionality. Data compression algorithms are very challenging for optimization on modern platforms due to strong data dependency (i.e., the behavior of the algorithm depends on the input data). Basically, it means that only a few new CPU instructions can be used to speed up these algorithms. The single instruction, multiple data (SIMD) instructions can be used here in limited cases, mostly pattern searching.

Performance optimization of data compression in Intel IPP takes place in different ways and at different levels:

- Algorithmic optimization
- Data optimization
- New CPU instructions
- Intel IPP data compression performance

Algorithmic Optimization

At the highest level, algorithmic optimization provides the greatest benefit. The data compression algorithms are implemented from scratch in Intel IPP to process input data and to generate output in the most effective way. For example, selecting proper starting conditions for input data processing can bring a performance gain from the very beginning. Let's look at the operation of searching for patterns in input data. If you set a minimum match length of two to three bytes, you will detect more matches—and obtain a better compression ratio. On the other hand, you will drastically increase input data processing time. So, the minimum match length should balance compression ratio and compression performance. This is mostly the result of statistical experiments and then careful planning of condition checks and branches in the code. In all cases, it is more effective to put checks for the most probable situations at the beginning of your code so that you will not have to spend time on conditional operations with a low probability of occurring.

Data Optimization

Careful planning of internal data layout and size also brings performance benefits. Proper alignment of data makes it possible to save CPU clocks on data reads and writes. In most CPUs, reading/writing executes faster on aligned data than on nonaligned data. This is especially true when memory operations are executed on data elements longer than 16 bits. Another factor affecting optimization is data size. If we keep internal data structures—arrays, tables, and lists—small, we can often reuse the CPU data cache, increasing overall performance.
New CPU Instructions
The Intel® Streaming SIMD Extensions 4.2 (Intel® SSE 4.2) architecture introduced several CPU instructions, which can be used in data compression algorithms according to their nature. More opportunities came with the Intel® Advanced Vector Extensions 2 instruction set. First, the CRC32 instruction can be used to compute the checksum of input data for data integrity. It is a key part of many data compression standards. The CRC32 instruction can also be used to implement a hash function, another key part of compression algorithms. Also, bit manipulation instructions (BMI) are used to speed up pattern matching functions. Finally, reading/writing data by 32-, 64-, 128-, or 256-bit portions saves memory bus cycles, which also improves performance.

Intel IPP Data Compression Performance
Each Intel IPP function includes multiple code paths, with each path optimized for specific generations of Intel® and compatible processors. Also, new optimized code paths are added before future processors are released, so developers can just link to the newest version of Intel IPP to take advantage of the latest processor architectures.

Figure 2 shows performance results with the Intel IPP ZLIB compression/decompression library. Compared with the open-source ZLIB code, the Intel IPP ZLIB implementation can achieve a 1.3x to 3.3x performance improvement at different compression levels. At the same time, Intel IPP provides fully compatible APIs and compression results.

![Performance Results Graph]

Configuration info – Software versions: Intel® Integrated Performance Primitives (Intel® IPP) 2017, Intel® C++ Compiler 16.0. Hardware: Intel® Core™ processor i7-6700K, 8 MB cache, 4.2 GHz, 16 GB RAM, Windows Server* 2012 R2. Software and workloads used in performance tests may have been optimized for performance only on Intel® microprocessors.

2 Performance results with the Intel® IPP ZLIB* compression and decompression library

For more complete information about compiler optimizations, see our Optimization Notice.
Choosing the Compression Algorithms for Applications

Since data compression is a trade-off between compression performance and compression ratio, there is no “best” compression algorithm for all applications. For example, the BZIP2 algorithm achieves good compression efficiencies but because it is more complex, it requires significantly more CPU time for both compression and decompression.

Figure 3 is a summary of the common Intel IPP functions on compression performance and ratio. Among these algorithms, the LZ4 is the fastest compression method—about 20x faster than BZIP2 on the standard corps compression data. But BZIPS could achieve the best compression ratios. ZLIB provides a balance between performance and efficiency, so the algorithm is chosen by many file archivers, compressed network protocols, file systems, and lossless image formats.

3 Common Intel® IPP functions and compression ratio

How can a user’s application decide the compression algorithm? Often, this involves a number of factors:

- **Compression performance** to meet the application’s requirement
- **Compression ratio**
- **Compression latency**, which is especially important in some real-time applications
- **Memory footprint** to meet some embedded target application requirements
- **Conformance with standard compression methods**, enabling the data to be decompressed by any standard archive utility
- **Other factors** as required by the application

Deciding on the compression algorithms for specific applications requires balancing all of these factors, and also considering the workload characteristics and requirements. There is no “best” compression method for all scenarios.
Consider a typical data compression usage example. Suppose we have a file download scenario that requires transferring data from servers to a client application. To maximize performance, lossless data compression is performed on the servers before transmission to the client. In this situation, the client application needs to download the data over the network and then decompress it. Because overall performance depends on both transmission and decompression performance, we need to carefully consider workload and network characteristics. Some high-compression algorithms, like BZIP2, could reduce data transmission. On the other hand, BZIP2 decompression takes longer to compute than other, less sophisticated methods.

**Figure 4** shows the overall performance of the common Intel IPP compression algorithms at different network speeds. Tests were run on the Intel® Core™ i5-4300U processor. When the code runs on some lower-bandwidth networks (e.g., 1 Mbps), an algorithm that can achieve high compression ratios is important to reduce the data/time for transmission. In our test scenario, Intel IPP BZIP2 achieved the best performance (approximately 2.7x speedup) compared to noncompressed transmission. However, when the data is transferred by a higher-performance network (e.g., 512 Mbps), LZ4 had the best overall performance, since the algorithm is better balanced for compression and transmission time in this scenario.

![Graph showing data transfer with and without compression](image)


Software and workloads used in performance tests may have been optimized for performance only on Intel® microprocessors.
Optimized Data Compression

Data storage and management are becoming high priorities for today’s data centers and edge-connected devices. Intel IPP offers highly optimized, easy-to-use functions for data compression. New CPU instructions and an improved implementation algorithm enable the data compression algorithms to provide significant performance gains for a wide range of applications and architectures. Intel IPP also offers common lossless compression algorithms to serve the needs of different applications.

---

**BLOG HIGHLIGHTS**

**How to Use the MPI-3 Shared Memory in Intel® Xeon Phi™ Processors**

**BY LOC Q NGUYEN (INTEL)**

MPI-3 shared memory is a feature introduced in version 3.0 of the message passing interface (MPI) standard. It is implemented in Intel® MPI Library version 5.0.2 and beyond. MPI-3 shared memory allows multiple MPI processes to allocate and have access to the shared memory in a compute node. For applications that require multiple MPI processes to exchange huge local data, this feature reduces the memory footprint and can improve performance significantly.

This white paper introduces the MPI-3 shared memory feature, the corresponding APIs, and a sample program to illustrate the use of MPI-3 shared memory in the Intel® Xeon Phi™ processor.
WRITE CODE THAT'S SCALABLE, PORTABLE, AND—BEST OF ALL—FUTURE-PROOF

Intel® Fortran and C++ Compilers optimize and accelerate code right from the development environment you already work in.

Write tomorrow's code today

Download for C++ ›

Download for Fortran ›

For more complete information about compiler optimizations, see our Optimization Notice.
Intel and the Intel logo 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
LeCloud is a leading video cloud service provider that deploys its content delivery network (CDN) nodes in more than 60 countries and regions, with 20 terabytes per second (Tbps) of available bandwidth to support its cloud services—including cloud-on-demand, content sharing, distribution and live broadcasting of commercial events, virtual reality, and more. LeCloud platforms support more than a million live events per year, hundreds of millions of device accesses a day, and millions of concurrent users per second. With the heavy demand for its high-performance public database services, LeCloud uses MySQL®, a popular open-source database, as the basis for its cloud service.
One key challenge for LeCloud is optimizing MySQL database performance―also an important topic for many database administrators (DBA) and operation and maintenance IT developers. Key performance concerns include:

- Queries per second (QPS)
- Transactions per second (TPS)
- Query response time

This article shows how we optimized MySQL using Intel® C++ Compiler and its Interprocedural Optimization (IPO) capability. Without having to modify any code, we compiled the MySQL source code and tested the performance indicators. The result? By using Intel C++ Compiler optimizations on MySQL in a set of test scenarios, performance improved between 5 and 35 percent―bringing developers another avenue to better MySQL performance.

**Intel C++ Compiler**

Intel C++ Compiler, a component of Intel® Parallel Studio XE, is a C and C++ optimizing compiler that takes advantage of the latest instruction sets and architectural features to maximize performance. Because the Intel® compiler development team knows the Intel® architecture so well, they can do specialized—and more effective—optimization for CPU features like new SIMD instructions and cache structure for Intel® CPUs compared to other compilers like GNU GCC* or Microsoft Visual C++*. Some of the optimization technologies used in the present case study include:

- Automatic vectorization
- Guided auto parallelism
- Interprocedural Optimization
- Profile-guided optimization

The Intel compilers help users get the most benefit out of their Intel®-based platforms.

**Automatic Vectorization**

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel compiler that automatically uses SIMD instructions in the Intel® Streaming SIMD Extensions (Intel® SSE, Intel® SSE2, Intel® SSE3, and Intel® SSE4), Supplemental Streaming SIMD Extensions (SSSE3) instruction sets, and the Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2, Intel® AVX512) instruction sets. The vectorizer detects operations in the program that can be done in parallel and converts the sequential operations to parallel. For example, the vectorizer converts the sequential SIMD instruction that processes up to 16 elements into a parallel operation, depending on the data type. The compiler also supports a variety of auto-vectorizing hints that can help the compiler generate effective vector instructions on the latest processors, including the Intel® Xeon® E5-2699 v4 used in this case study.
Guided Auto Parallelism

The Guided Auto Parallelism (GAP) feature of the Intel C++ Compiler offers advice to improve the performance of sequential applications by suggesting changes that will take advantage of the compiler's ability to automatically vectorize and parallelize code as well as improve the efficiency of data operations.

Interprocedural Optimization (IPO)

This automatic, multistep process allows the compiler to analyze code to find interprocedural optimizations (i.e., optimizations that go beyond individual program subunits) within source files and across multiple source files. IPO is covered in more detail below, as it is a key component of this case study.

Profile-Guided Optimization (PGO)

In PGO, the compiler analyzes the code while it runs and takes advantage of this profile information during subsequent recompilation. This improves application performance by shrinking code size, reducing branch mispredictions, and reorganizing code layout to minimize instruction cache problems.

MySQL Optimization

Compiling with IPO

IPO is a key optimization technique in the Intel C++ Compiler. It does code profiling and static topological analysis based on both single-source and multisource files, and then implements specific optimizations like inlining, constant propagation, and dead function elimination for programs that contain many commonly used small- and medium-sized functions. Figure 1 shows the IPO process.

When you compile your source code with the IPO option, for single-file compilation, the compiler performs inline function expansion for calls to procedures defined within the current source file. For multifile IPO, the compiler may perform some inlining according to the multisource code, such as inlining functions marked with inlining pragmas or attributes (GNU C and C++) and C++ class member functions with bodies included in the class declaration. After each source file is compiled with IPO, the compiler stores an intermediate representation of the source code in mock object files.

When you link with the IPO option, the compiler is invoked a final time to perform IPO across all mock object files. During the analysis process, the compiler reads all intermediate representations in the mock file, object files, and library files to determine if all references are resolved and whether a given symbol is defined in a mock object file. Symbols included in the intermediate representation in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

For more complete information about compiler optimizations, see our Optimization Notice.
Building MySQL with the Intel Compiler

The latest official version of MySQL (v5.6.27) was used in this case study. This section describes how to use Intel C++ Compiler to compile MySQL on Linux*.

Download the MySQL installation package:

```
wget http://downloads.mysql.com/archives/get/file/mysql-5.6.27.tar.gz
tar –zxvf mysql-5.6.27.tar.gz
```

- Compile MySQL with Intel C++ Compiler (Figure 2):
  1. Install Cmake: `yum -y install wget make cmake gcc gcc-c++ autoconf automake zlib* libxml2* ncurses-devel libmcrypt* libtool-ltdl-devel*`
  2. Compile MySQL with Intel C++ Compiler: Set CC to icc, CXX to icpc, and enable IPO with the –ipo option.
Compiling MySQL with Intel® C++ Compiler

- Create MySQL Grant System Tables
  
  ```
  cd /usr/local/mysql-5.6.27-icc
  groupadd mysql
  useradd -M -g mysql mysql -s /sbin/nologin ;
  chown -R mysql .
  chgrp -R mysql .
  ./scripts/mysql_install_db --user=mysql --collation-server=utf8_general_ci
  ```

Performance Testing

The purpose of this analysis is to do comparative performance testing between MySQL built with the GNU and Intel compilers. The performance metrics for online transaction processing (OLTP) include queries per second (QPS) and response time (RT). The test tool was Sysbench® (v0.4.12), a modular, cross-platform, multithreaded benchmarking tool that is commonly used to evaluate database performance. Tables 1 and 2 show the test environment.
Table 1. Hardware environment

<table>
<thead>
<tr>
<th>Hardware Environment</th>
<th>Server</th>
<th>Dell PowerEdge* R730xd</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel® Xeon® processor E5-2699 v4 @ 2.20 GHz 2</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>128 GB</td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td>SATA* SSD S3510, 1.5T 1</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Software environment

<table>
<thead>
<tr>
<th>Software Environment</th>
<th>OS version</th>
<th>CentOS* 6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>2.6.32-926</td>
<td></td>
</tr>
<tr>
<td>Intel® C++ Compiler</td>
<td>ICC-17.0.1</td>
<td></td>
</tr>
<tr>
<td>GNU GCC*</td>
<td>4.4.7 20120313 (RedHat* 4.4.7-11)</td>
<td></td>
</tr>
<tr>
<td>MySQL*</td>
<td>MySQL 5.6.27</td>
<td></td>
</tr>
<tr>
<td>Sysbench*</td>
<td>Sysbench-0.4.12</td>
<td></td>
</tr>
</tbody>
</table>

Test Steps

We installed two MySQL instances, 3308 and 3318, on the same Intel® SATA* SSD disk and used Sysbench for OLTP testing (Table 3). Before testing, we cleaned the operating system cache and disabled the MySQL query_cache. We created 10 MySQL database tables for testing, with one million records per table. 4, 8, 16, 32, 64, 128, and 512 test threads were used for random read OLTP. MySQL buffer_pool size was set as 2 GB, 8 GB, and 16 GB. Each test was run three times and the average time was used as the final result.
Table 3. Test steps

<table>
<thead>
<tr>
<th>Instance</th>
<th>MySQL* Version</th>
<th>Compiler</th>
<th>Engine</th>
<th>QueryCache</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL 3308</td>
<td>5.6.27</td>
<td>GCC*</td>
<td>Innodb</td>
<td>Disable</td>
</tr>
<tr>
<td>MySQL 3318</td>
<td>5.6.27</td>
<td>Intel® C++ Compiler</td>
<td>Innodb</td>
<td>Disable</td>
</tr>
</tbody>
</table>

Performance Testing Results

QPS

The results of the Sysbench TPS test on MySQL compiled with the Intel and GNU compilers are shown in Table 4 and Figure 3. The number of threads and the MySQL buffer_pool size were varied. Across the board, MySQL compiled with the Intel compiler gives significantly higher QPS, especially as the number of threads increases. We see a 5 to 35 percent improvement in QPS for MySQL compiled with the Intel C++ Compiler (Figure 3).

Table 4. Queries per second (higher is better)

<table>
<thead>
<tr>
<th>Threads</th>
<th>GCC_2GB</th>
<th>ICC_2GB</th>
<th>GCC_8GB</th>
<th>ICC_8GB</th>
<th>GCC_16GB</th>
<th>ICC_16GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1036.64</td>
<td>1184.88</td>
<td>1406.89</td>
<td>1598.23</td>
<td>1402.00</td>
<td>1592.23</td>
</tr>
<tr>
<td>8</td>
<td>1863.62</td>
<td>2154.04</td>
<td>2556.77</td>
<td>2882.07</td>
<td>2541.71</td>
<td>2856.27</td>
</tr>
<tr>
<td>16</td>
<td>3394.35</td>
<td>4013.79</td>
<td>4949.17</td>
<td>5444.82</td>
<td>4915.52</td>
<td>5468.59</td>
</tr>
<tr>
<td>32</td>
<td>5639.77</td>
<td>6625.03</td>
<td>9036.91</td>
<td>9599.00</td>
<td>9034.98</td>
<td>9570.54</td>
</tr>
<tr>
<td>64</td>
<td>6975.84</td>
<td>9510.31</td>
<td>12796.34</td>
<td>14680.04</td>
<td>12839.79</td>
<td>14662.85</td>
</tr>
<tr>
<td>128</td>
<td>6975.05</td>
<td>9465.03</td>
<td>12410.70</td>
<td>14804.35</td>
<td>12391.96</td>
<td>14744.11</td>
</tr>
<tr>
<td>256</td>
<td>6859.37</td>
<td>9367.19</td>
<td>11586.47</td>
<td>14083.09</td>
<td>11574.40</td>
<td>13887.13</td>
</tr>
<tr>
<td>512</td>
<td>6769.54</td>
<td>9260.24</td>
<td>10904.25</td>
<td>12290.89</td>
<td>10893.18</td>
<td>12310.28</td>
</tr>
</tbody>
</table>

For more complete information about compiler optimizations, see our Optimization Notice.
Average Response Time

The results of the Sysbench average response time (RT) test on MySQL compiled with the Intel and GNU compilers are shown in Table 5 and Figure 4. RT is in milliseconds. The number of threads and the MySQL buffer_pool size were varied as in the QPS test. Once again, MySQL compiled with the Intel compiler gives superior performance, especially at higher numbers of threads.

Table 5. Average response time (in milliseconds, lower is better)

<table>
<thead>
<tr>
<th>Threads</th>
<th>GCC_2GB</th>
<th>ICC_2GB</th>
<th>GCC_8GB</th>
<th>ICC_8GB</th>
<th>GCC_16GB</th>
<th>ICC_16GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.86</td>
<td>3.38</td>
<td>2.84</td>
<td>2.50</td>
<td>2.85</td>
<td>2.51</td>
</tr>
<tr>
<td>8</td>
<td>4.29</td>
<td>3.71</td>
<td>3.13</td>
<td>2.77</td>
<td>3.15</td>
<td>2.80</td>
</tr>
<tr>
<td>16</td>
<td>4.71</td>
<td>3.98</td>
<td>3.23</td>
<td>2.94</td>
<td>3.25</td>
<td>2.93</td>
</tr>
<tr>
<td>32</td>
<td>5.67</td>
<td>4.83</td>
<td>3.54</td>
<td>3.33</td>
<td>3.54</td>
<td>3.34</td>
</tr>
<tr>
<td>64</td>
<td>9.17</td>
<td>6.73</td>
<td>5.00</td>
<td>4.36</td>
<td>4.98</td>
<td>4.36</td>
</tr>
<tr>
<td>128</td>
<td>18.35</td>
<td>13.52</td>
<td>10.31</td>
<td>8.64</td>
<td>10.33</td>
<td>8.68</td>
</tr>
<tr>
<td>256</td>
<td>37.32</td>
<td>27.33</td>
<td>22.09</td>
<td>18.18</td>
<td>22.12</td>
<td>18.45</td>
</tr>
<tr>
<td>512</td>
<td>75.63</td>
<td>55.29</td>
<td>46.95</td>
<td>41.65</td>
<td>47.00</td>
<td>41.59</td>
</tr>
</tbody>
</table>
3 Queries per second for Intel® C++ Compiler versus GNU GCC*

4 Response time, Intel® C++ Compiler versus GNU GCC*
Conclusions

Maximizing MySQL performance is essential for DBAs, operators, and developers trying to reach a performance goal. This study demonstrates that compiling MySQL with the Intel C++ Compiler can significantly improve database performance. We found that the Intel C++ Compiler improved performance by 5 to 35 percent, with an average improvement of about 15 percent.

There are many factors that affect MySQL performance, including the MySQL configuration, the CPU, and the SSD. For example, the Intel®Xeon® processor E5-2620 v3 was used initially, but upgrading to the Intel®Xeon processor E5-2699 v4 improved performance. Using a faster SSD, such as the Intel® DC P3700 instead of the Intel® DC S3510 used in this article, should also further improve performance.

References

1. Intel® Media Server Studio Support
3. CMake Reference Documentation
ACCELERATING LINEAR REGRESSION IN R* WITH INTEL® DAAL

Make Better Predictions with This Highly Optimized Open Source Package

Steena Monteiro, Software Engineer, Intel Corporation, and Shaojuan Zhu, Technical Consulting Engineer, Intel Corporation

Linear regression is a classic statistical algorithm that is used to predict an outcome by leveraging information from previously collected observations. Several businesses make predictions on previously collected data and monetize on the results; common examples include forecasting sales numbers in a new quarter and predicting credit risk given spending patterns. Intel® Data Analytics Acceleration Library (Intel® DAAL) contains a repertoire of tuned machine learning algorithms optimized for use on Intel® processors. In this article, we leverage the power of tuned linear regression in Intel DAAL to accelerate linear regression in R*, a software language for statistical algorithms and analysis. A preliminary performance analysis indicates that linear regression in Intel DAAL shows better system utilization with higher arithmetic intensity. Our
experiments demonstrate that incorporating Intel DAAL in linear regression in R speeds up model training by up to 300x, and speeds up model inference by up to 373x.

**Overview of Linear Regression**

Linear regression\(^1\) is a classic statistical algorithm used for predicting values of a dependent variable. Simple linear regression predicts a single response (dependent variable, \(y\)) using a single independent variable (\(x\)) and takes the following form:

\[ y = \beta x + \epsilon \]

where \(\beta\) is the coefficient (slope) of the regression line, and \(\epsilon\) is the intercept.

Given a set of points, the goal of a linear regression model is to derive a line that will minimize the squared error of the response variable. Real-world data sets contain multiple independent variables (or features) to describe the domain they represent. An extension of simple linear regression—multiple linear regression—is used to represent a multidimensional problem space. Multiple linear regression is represented as:

\[ y = \alpha x_1 + \beta x_2 + \gamma x_3 + \ldots + \eta x_n + \epsilon \]

where \(x_i\) are feature variables, and \(\alpha, \beta, \gamma, \) and \(\eta\) are the corresponding coefficients.

In this article, we use linear regression to mean multiple linear regression.

**Analyzing Arithmetic Intensity of Linear Regression in Intel DAAL and R**

An algorithm's utilization of a system's floating point and bandwidth capability can be evaluated by calculating the application's arithmetic intensity. Arithmetic intensity\(^2\) is defined as the ratio of floating point operations (FLOPs) performed by the application to the bytes used during the application's execution. Arithmetic intensity that is close to the theoretical arithmetic intensity of the system indicates better system utilization. Given the high-performance potential of Intel DAAL, we examine the arithmetic intensity of linear regression in R and Intel DAAL through the Intel® Software Development Emulator (Intel® SDE).\(^3\) We measure FLOPs and bytes across three data sets synthesized to fit into L1 cache (32K), L2 cache (256K), and L3 cache (20 MB). (Table 1 shows the specs of the node used in our experiments.) Based on our experiments, the arithmetic intensity of linear regression in Intel DAAL is 2.4x, 1.4x, and 1.4x greater than that of linear regression in R, across the three data sets. Encouraged by these numbers, we proceed with improving performance of linear regression in R using Intel DAAL.
Table 1: Haswell node specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>8 per socket</td>
</tr>
<tr>
<td>Sockets</td>
<td>1</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.30 GHz (3.6 GHz Turbo)</td>
</tr>
<tr>
<td>L1</td>
<td>32K</td>
</tr>
<tr>
<td>L2</td>
<td>256K</td>
</tr>
<tr>
<td>L3</td>
<td>20480K</td>
</tr>
</tbody>
</table>

Table 2: Linear regression data sets from the University of California–Irvine (UCI) machine learning repository

<table>
<thead>
<tr>
<th>Data set</th>
<th>Rows (instances)</th>
<th>Columns (features)</th>
<th>Size (MB)</th>
<th>Predicted response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airfoil self-noise</td>
<td>1503</td>
<td>5</td>
<td>0.06</td>
<td>Sound pressure level</td>
</tr>
<tr>
<td>Parkinsons</td>
<td>5875</td>
<td>19</td>
<td>0.83</td>
<td>Clinician's Parkinson's disease symptom score</td>
</tr>
<tr>
<td>Naval propulsion plants</td>
<td>11934</td>
<td>13</td>
<td>1.24</td>
<td>Gas turbine compressor decay state coefficient</td>
</tr>
<tr>
<td>Poker</td>
<td>1000000</td>
<td>10</td>
<td>80</td>
<td>Card suit</td>
</tr>
<tr>
<td>Year prediction million songs database</td>
<td>515345</td>
<td>90</td>
<td>371.04</td>
<td>Prediction of song release year</td>
</tr>
</tbody>
</table>

Linear Regression API in R

R is a software environment and a language that comes with a host of prepackaged algorithms for data manipulation, classical statistics, graphics, bioinformatics, and machine learning.\(^5\) R provides linear regression through the `lm` function.\(^6\) The `lm` function is used to train a linear regression model, and a `predict` function is used to make inferences on new data.
Linear Regression in Intel DAAL

Linear regression is one of the two regression algorithms in Intel DAAL. Linear regression in Intel DAAL can be executed in three different data processing modes—batch, online, and distributed. The batch mode can be thought of as a serial mode of implementation, where all the data is processed sequentially. In the online mode, the algorithm trains data in sets of blocks as data becomes available. In the distributed mode, linear regression in Intel DAAL can be trained and tested across multiple compute nodes. Intel DAAL provides a `compute` method through the `Algorithm` class that is used for linear regression training and testing. In its native form, linear regression in R can only be executed in serial. For very big data sets that do not fit into memory, R will not work. However, Intel DAAL makes it possible to work on large data sets through its online or streaming mode. For this article, we use linear regression in batch mode to accelerate the algorithm in R. We use the QR method for linear regression in both R and Intel DAAL.

Integrating Linear Regression in R Using Intel DAAL

We use Intel DAAL’s C++ API for our experiments on linear regression. The Rcpp package in R provides a way to integrate C++ functionality into native R code. Rcpp enables data transfer between R objects and Intel DAAL C++ code. To successfully integrate Intel DAAL with R’s linear regression, we wrote Intel DAAL wrappers in C++ to load data from a file, train, and test a linear regression model. For details on implementing these wrappers, please refer to Lightning-Fast Machine Learning R* Algorithms.

To monitor performance benefits, we time the `lm` and `predict` function for linear regression training and testing, respectively, in native R. For the Intel DAAL wrapper, we measure Intel DAAL’s `compute` execution that is the core of Intel DAAL’s linear regression training and testing.

Table 2 provides a brief description of the data sets we used in this study. We use 75 percent of each data set for linear regression training, and the remaining 25 percent for testing.

Dimension Reduction: Detecting and Eliminating Multicollinearity

A stable linear regression model requires that predictive features in the data should be free from multicollinearity. Features are said to be multicollinear when sets of features are highly correlated and can be expressed as linear combinations of each other. Ignoring multicollinearity can result in a model that makes misleading predictions. We carry out analysis to detect multicollinearity on features in the naval propulsion plants data set. Intel DAAL provides a set of distance functions, namely, `correlation_distance` and `cosine_distance` to measure correlations between different feature variables. While exploring the data set, we noticed two features—starboard propeller torque and port propeller torque—to be duplicates of each other. We use box plots to summarize the distribution of values inside these features. Figure 1 shows the minimum, lower quartile, mean, upper quartile, and maximum values of both torques. The box plots are
replicas of each other. This means we can eliminate one of these features, since they are highly correlated and their duplication does not produce additional insight. While examining distributions of data across the features, we noticed two features—gas turbine compressor inlet and compressor outlet pressure—to be uniformly distributed (Figures 2 and 3) and consequently produce an unstable linear regression model. We thus eliminated three out of the 16 features to train and test linear regression on the naval propulsion plant data set. As a side note, another alternative would be to use ridge regression (included in Intel DAAL) due to its capability of working with collinear predictors.

![Distribution of multicollinear features in the naval propulsion plant's data set](image)
2. Uniform distribution of air pressure in the naval propulsion plant's data set

3. Uniform distribution of air pressure in the naval propulsion plant's data set
Analyzing Speedup in Native R by Using Intel DAAL

We executed linear regression on a single-socket, eight-core node (Table 1). To eliminate effects of possible system noise, we executed training and testing experiments on every data set 10 times. Figure 4 shows training speedup between linear regression in R and Intel DAAL-enhanced R, and Figure 5 shows testing speedup.

There is a significant training and testing speedup across all data sets, as seen in Figures 4 and 5. However, there is also a decrease in performance benefit from using Intel DAAL with R in data sets that are smaller than L3 cache capacity. Training speedup almost plateaus for larger data sets, partly because performance of linear regression becomes memory limited for larger data sets. This is also supported by the lower arithmetic intensity in larger data sets in our preliminary performance analysis. Because we cannot leverage Intel DAAL’s sophisticated data-management in the batch (serial) mode, we do not see a higher speedup on training and testing data sets that exceed L3 cache capacity.

![Speedup from using DAAL for linear regression training in R](chart.png)

4 Linear regression training speedup
Conclusion

Linear regression is a common statistical algorithm used for making predictions. Several open source packages provide functions to carry out linear regression. Intel DAAL is a highly optimized data analytics library for Intel® architectures that contains tuned implementations of many statistical machine learning algorithms. We successfully leverage Intel DAAL to speed up linear regression in R. Our experiments show a training speedup up to 300x and a testing speedup up to 373x.
References
1. Introduction to linear regression.
2. Arithmetic intensity and the roofline model.
3. Intel® Software Development Emulator.
4. UCI machine learning repository.
6. Linear regression in R*.
7. Linear regression in DAAL.
8. Rcpp: Seamless R* and C++ Integration.
10. Introduction to box plots.

BLOG HIGHLIGHTS

Exploring MPI for Python* on Intel® Xeon Phi™ Processors
BY LOC Q NGUYEN (INTEL) ›

Message Passing Interface (MPI) is a standardized message-passing library interface designed for distributed memory programming. MPI is widely used in the high performance computing (HPC) domain because it is well suited for distributed memory architectures.

Python* is a modern, powerful interpreter that supports modules and packages. Python supports extension C/C++. While HPC applications are usually written in C or Fortran for faster speed, Python can be used to quickly prototype a proof of concept and for rapid application development because of its simplicity and modularity support.

The MPI for Python (mpi4py) package provides Python bindings for the MPI standard.

Read more