Boosting Speech Recognition Performance by 5x

Intel® Math Kernel Library, Intel® C++ Compiler

High-Performance Computing

Qihoo360 Technology Co. Ltd. Optimizes its Applications on Intel® Architecture

The Euler* platform is an important distributed offline computing platform that supports machine-learning-related computation models for real businesses. For Qihoo360 Technology Co. Ltd., a Chinese Internet security company, the performance of the Euler platform on Intel® Xeon® processor-based servers is critical. The company collaborated with Intel to optimize its applications on Intel® architecture. Engineers from both companies worked together closely to optimize the speech recognition module of the Euler platform using Intel® Math Kernel Library (Intel® MKL), speeding up performance by 5x.

Business Requirements

Qihoo360 has about 500 million monthly active Internet users and over 640 million mobile users. Recognizing security as a fundamental need of all Internet and mobile users, the company built its user base by offering comprehensive, effective, and user-friendly Internet and mobile security products and services to protect users’ computers and mobile devices against malware and malicious websites. Besides its 360 Safe Guard* and 360 Internet Security* products, Qihoo360 has also released new products, including 360 Cloud*, 360 Browser*, 360 Search*, and 360 Mobile Assistant*.

In the mobile Internet era, with a rapidly growing business and user base, Qihoo360 faces growing pressure to support its increasing user base and business and reduce the total cost of ownership. In Qihoo360's data center, most services and applications run on the Intel architecture platform. And it has become increasingly urgent to improve their performance and make them run efficiently on Intel architecture.

Qihoo360 Euler Overview

Qihoo360's Euler platform is a distributed offline computing platform for large-scale data processing that adopts mainly a full in-memory computation model. The Euler platform is designed to efficiently handle computation models that require multiple iterations, such as machine learning. Machine learning is a scientific
“We are glad that we engaged with Intel to apply Intel® MKL and Intel® C/C++ Compiler to improve the computing speed of our Euler ASR module on Intel® architecture. This will help us to maintain our tech leadership in the industry and to grasp new opportunities to grow our business.”

—Dr. Bai Ming Architect Qihoo360 Technology Co. Ltd.

Maintaining Tech Leadership and Maximizing New Growth Opportunities

The Euler platform serves as the offline training model for Qihoo360’s business, including its click-through rate estimate model, page ranking, white listing, mobile assistant recommendations, image similarity computation in image searches, network attack behavior analysis model, word2vec model, and speech recognition. In the Euler platform, input and output (I/O) data are from a distributed file system (such as HDFS*). Besides this, there is no other I/O overhead, making the Euler platform a typical computation-intensive workload.

Improving the computational performance of Euler on Intel Xeon processor-based servers is a big challenge for Qihoo360. The speed of training the machine learning model in the Euler platform (e.g., speech recognition model training) has a direct impact on Qihoo360’s business.

Engineers from both Qihoo360 and Intel worked closely together to optimize the key speech recognition module in Qihoo360’s Euler platform. Speech recognition is the translation of spoken words into text, also known as automatic speech recognition (ASR). Speech recognition is used by Qihoo360’s voice search service.

Euler provides an API and library (C++ only) and framework to Qihoo360’s service developers, who can then focus on machine-learning-related usage development.
Linux*, OS X*, and Android*), including Intel® C/C++ Compiler and Intel® MKL.

Intel® C/C++ Compiler Overview

Intel engineers designed Intel C/C++ Compiler for leading performance on Intel and compatible processors. The compiler has unique capabilities to take advantage of the new multicore processors and multiprocessor systems. It automatically generates optimized code for all Intel architecture-based platforms including automatic vectoring and parallelizing, memory and cache line tuning, as well as serious high-level optimization. It explores how to complete a task with minimal CPU cycles and is compatible with Microsoft Visual C++* on Windows* and GNU Compiler Collection* (GCC*) on Linux*. It also scales forward with multicore, many-core, and multiprocessor systems with OpenMP, automatic parallelism, and Intel® Xeon Phi™ coprocessor support. All compiler features help developers cut development time, costs, and risk, as well as to achieve the best possible performance on Intel architecture.

Intel® MKL Overview

Intel MKL (Figure 3) is a feature-rich, high-performance mathematical library for building scientific, engineering, and financial applications. It gives developers better performance with minimal effort. Intel MKL is optimized for the latest Intel® processors, including Intel's multicore and many-core processors. It can unleash the maximum computing and parallel processing capability of the Intel Xeon and Xeon Phi product families. (See the Intel MKL release notes for the full list of supported processors.)

Intel® MKL provides rich functions, including Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) routines, Fast Fourier transforms, vector math functions, random number generation functions, and other functionality. It also provides interfaces to de facto standard APIs including C++, Fortran, C#, Java, Python, and R.

Speech Recognition Optimization

ASR is a typical large-scale, compute-intensive module in Qihoo360’s Euler platform. Deep Neural Networks (DNN) was used for the task. As Figure 2 shows, after preprocessing a massive amount of speech data, all floating-point data were fed into the DNN. After repeated iterations, an acoustic model was established. During processing, there were huge matrix and vector operations that became a bottleneck for the whole project.

Computation speed became a bottleneck in Qihoo360’s ASR module, but simply expanding the server pool was not a cost-effective solution. The company needed an optimized software solution.

Intel® Parallel Studio XE is a suite of software development tools that can maximize Intel architecture capabilities and make it easier to deploy advanced technologies. It includes a full set of high-performance libraries for all Intel architecture-based client and server platforms for multiple operating systems (e.g., Windows*, Linux*, OS X*, and Android*), including Intel® C/C++ Compiler and Intel® MKL.

Table 1. CBLAS functions in DNN modeling

<table>
<thead>
<tr>
<th>Operation</th>
<th>CBLAS Routine “S is for Single Floating”</th>
<th>Example</th>
<th>Computational Complexity (Work)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Scalar</td>
<td>cblas_SSCAL</td>
<td>x = a\cdot x</td>
<td>O(N)</td>
</tr>
<tr>
<td>Vector Vector</td>
<td>cblas_SAXPY</td>
<td>y = y + a \cdot x</td>
<td>O(N)</td>
</tr>
<tr>
<td>Matrix Vector</td>
<td>cblas_SGERM</td>
<td>y = a\cdot\text{Ax} + \text{By}</td>
<td>O(N^p)</td>
</tr>
<tr>
<td>Matrix Matrix</td>
<td>cblas_SGEMM</td>
<td>C = a\cdot A\cdot B + D\cdot C</td>
<td>O(N^p)</td>
</tr>
</tbody>
</table>

Figure 2. Large vocabulary speech recognition in the Euler platform

Figure 3. Intel® Math Kernel Library

Table 1. CBLAS functions in DNN modeling
Architecturally, the Intel MKL BLAS library is optimized with Single Instruction Multiple Data instructions such as Advanced Vector Extensions (AVX and AVX2). With multi-threaded features, it can be used as a performance building block in many math libraries and applications. BLAS itself is a popular math computing routine set and a de facto standard API for linear algebra libraries. It performs common linear algebra operations such as vector scaling, vector dot product, matrix vector multiplication, and matrix multiplication (Table 1).

BLAS was first implemented as a Fortran library in 1979 by netlib. There are several variations and alternative implementations such as CBLAS* (C interface to the BLAS), highly optimized implementations such as Intel MKL and AMD ACML*, as well as GotoBLAS* and ATLAS* (a portable, self-optimizing BLAS). BLAS libraries are widely used as building blocks in high-level math programming languages and libraries including LINPACK*, LAPACK*, MATLAB*, GNU Octave*, Mathematica*, NumPy*, and R* and in all kinds of computational applications. Developers can easily replace any BLAS library with the Intel MKL BLAS library and speed the BLAS computation on Intel architecture. The Intel MKL BLAS library can be integrated with many third-party libraries including MATLAB*, NumPy*, Scipy* and R, Armadillo*, and Boost* uBLAS.

Intel CC and Intel MKL Based Solution
As explained earlier, Qihoo360 needed a way to optimize its compute-intensive application. Originally, Qihoo360 used the ATLAS CBLAS library as the basic component in its computation layer with GCC as the default compiler. The CBLAS library is the C interface of the BLAS library, so it is often used as a basic computing building block to support the upper-level application and has a de facto standard API. A developer can change the CBLAS library to an optimized BLAS library without changing any code. Intel MKL provides the BLAS library and the CBLAS library, which have the same API as a standard CBLAS library. The Intel Compiler can extract top performance on Intel architecture, so Qihoo360’s Euler team sought to replace the GCC compiler and CBLAS library with the Intel Compiler and Intel MKL to build the ASR module.

As Figure 2 shows, DNNs are the most important part of the ASR module. It is constructed by an extensive linear algebra operation, which is supported by the CBLAS API (Table 1).

We take the matrix multiply function cblas_sgemm. The code in the left column of Table 2 is the prototype of the functionality, while the right column displays the CBLAS implementation.

We built the function cblas_dgemm with ATLAS CBLAS along with the OS (from the Red Hat yum* repository) and the Intel MKL CBLAS library (as part of Intel Parallel Studio XE 2013 SP1). The original code is unchanged. The only thing we changed was the link line shown in Table 2, replacing the ATLAS CBLAS library with Intel MKL BLAS. Because the Intel MKL is optimized for new AVX instructions and well-tuned for multicore platforms, Intel MKL has overwhelmingly dominant performance compared to OS-associated ATLAS CBLAS, even with the multithreaded version of libptcblas.so. The first category in Figure 4 is the speedup result of cblas_sgemm using mkl blas comparing using the original ATLAS BLAS along with the OS. The performance metric is the speedup. We can see in Figure 4 that the Intel MKL BLAS is 7x faster than original ATLAS BLAS on our test machine (Intel® Xeon® processor E 5-2680 2.70GHz, 2 sockets and 8 cores, AVX support, Red Hat Enterprise Linux* Server release 6.3).

Furthermore, since Intel MKL is well-tuned for other Intel processors, when Qihoo360 upgrades to a new hardware solution like AVX2, it can automatically take advantage of the new hardware via the Intel MKL CBLAS library and keep seeing outstanding performance.

Besides the CBLAS functions, the ASR module also includes many vector operations. A common function func_12() is an example. It includes all kinds of operations like finding the maximum value of the vector, computing the exponent of one vector, and summing them. Originally, we tried to build the function with GCC and apply some CBLAS functions. We got some performance improvement, but no speedup since most

Table 2. Build function cblas_sgemm with ATLAS BLAS and Intel® MKL

<table>
<thead>
<tr>
<th>Algorithm C Prototype</th>
<th>CBLAS Implementation</th>
</tr>
</thead>
</table>
| void func_a_1(const int row, const int col, const float* const in_vec1, const float* const in_vec2, float* const out_vec)
| { Float sum; 
| for (int i = 0; i < row; i++)
| for (int j = 0; j < col; j++)
| sum += in_vec1[i] * col[j];
| out_vec[offset + i] = sum; } |
| void func_a_2(const int row, const int col, const float* const in_vec1, const float* const in_vec2, float* const out_vec) |
| { Float alpha = 1.0, beta = 0.0;
| cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, row, col, alpha, in_vec1, col, in_vec2, beta, out_vec, col); } |
| Build command line: |
| Open Source: $g++ -o2 test_sgemm.cc -I/CBLAS/include -I/scratch/scratch64/atlascblas.so.3 -openmp -lm -lpthread |
| Intel MKL: $g++ -o2 test_sgemm.cc -I/CBLAS/include -I/mkl/intel/64-bit -lmkl_intel_thread -lmkl_core -ltomp5 -lm -lpthread |
of them are of O(N) computational complexity and hard to multi-thread. Because the Intel Compiler can automatically generate optimized code, including autovectorization and parallelizing and memory and cache line tuning, building with the Intel Compiler improved performance 11x. (The second bar in Figure 4 shows this speedup.)

These test results inspired Qihoo360 to adopt the Intel Compiler and Intel MKL into the whole module. With the Intel MKL CBLAS library, Intel MKL Vector Math library, Intel Compiler, and some optimization techniques—including memory alignment, CPU KMP affinity, vectorization, and SSE instruction—ASR performance improved about 5x over the previous implementation (Figure 4).

**Summary**

Just replacing the default CBLAS with the Intel MKL CBLAS library let Qihoo360’s developers unleash the performance of Intel processors. Moreover, Intel MKL is optimized for Intel’s existing and future processors, so the application can help Qihoo360 automatically gain performance on a new processor without any code changes. At the same time, the Intel Compiler can help developers further improve performance easily and flexibly. With these tools, developers can achieve great performance speedup with less effort, improving the efficiency of their application development, saving maintenance time, and enjoying better scalability on Intel architecture.

Qihoo360 is satisfied with the results and plans to promote the solution with future development on other compute-intensive modules.

### Table 3. Build function `func_12` with GCC* and Intel® C++ Compiler

<table>
<thead>
<tr>
<th>Build function</th>
<th>GCC*</th>
<th>Intel® C++ Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>func_12</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 4. ASM optimization results

Download the free Community Licensed Version of Intel® MKL >