acmqueue

Returning Control to the Programmer: SIMD Intrinsic for Virtual Machines

Exposing SIMD units within interpreted languages could simplify programs and unleash floods of untapped processor power.

Jonathan Parri, Daniel Shapiro, Miodrag Bolic, and Voicu Groza

Server and workstation hardware architecture is continually improving, yet interpreted languages—most importantly, Java—have failed to keep pace with the proper utilization of modern processors. SIMD (single instruction, multiple data) units are available in nearly every current desktop and server processor and are greatly underutilized, especially with interpreted languages. If multicore processors continue their current growth pattern, interpreted-language performance will begin to fall behind, since current native compilers and languages offer better automated SIMD optimization and direct SIMD mapping support. As each core in commercial x86 multicore processors includes a dedicated SIMD unit, the performance disparity will grow exponentially as long as the available SIMD units remain underutilized in interpreted-language environments.

SIMD FOR EVERYONE

Software design and computer architecture have seen the evolution of parallel data processing, but it has not been fully utilized within the interpreted-language domain. Taking full advantage of the available system architecture and features, especially SIMD units, can be very challenging for the developer. Such features are often disregarded in favor of costly but scalable measures to increase performance, mainly the addition of processing power. Virtual machines and interpreted languages have consistently abstracted and moved away from the underlying system architecture.

SIMD instructions were added to modern processors to increase instruction-level parallelism. In a SIMD instruction, multiple and unique data elements can be manipulated with one common operation. SIMD units typically include an additional register bank with a large register width. These individual registers can be subdivided to hold multiple data elements of varying data types.

Developers have begun to take note that SIMD instruction capabilities are underutilized in the CPU.⁴ In interpreted languages, a portion of the software-to-hardware mapping occurs on the fly and in realtime, leaving little time for thorough instruction-level parallelism analysis and optimization. Bytecode compilation can be used to identify parallelism opportunities but has proven to be ineffective in many realistic scenarios. Exhaustive automated SIMD identification and vectorization is too computationally intensive to occur within JIT (just-in-time) compilers.

It is now common to find vector operations on arrays being performed in native languages with the help of SIMD-based instruction-set extensions (e.g., AltiVec, SSE, VIS), general-purpose computing on graphics cards (e.g., Nvidia CUDA, ATI STREAM), and memory modules. AltiVec, SSE, and VIS are well-known SIMD instruction sets. AltiVec is commonly found on PowerPC systems, SSE on x86-64, and VIS within SPARC.

We propose that generic user-known parallel operation calls be mapped directly in virtual machine internals to native SIMD calls. By allowing generic SIMD vector access within interpreted

languages, the user is given more control over parallelism, while still allowing for automated SIMD vectorization by the JIT compiler. Future virtual machine developments and language specifications must adapt to make better use of SIMD units. Language support for SIMD will not impact support for processors that do not contain SIMD operations. Instead the virtual machine must translate the generic calls into sequential instructions when no SIMD hardware is detected: the status quo.

Resource utilization is a key tenet of the cloud-computing paradigm that is now attracting many companies. Maximizing the utilization potential of cloud resources is vital to lowering cost and increasing performance. Because of their popularity and interoperability, interpreted languages, mainly Java, are frequently chosen for cloud computing. Interpreted languages within cloud environments do not expose processor architecture features to the programmer, and as a result, the generated code is rarely optimized for the target platform processor resources. More efficient use of existing data-center resources would surely result in cost reduction.

Interpreted languages such as Java, Flash, PHP, and JavaScript may all benefit from such an approach. There are many potential arguments for and against SIMD intrinsics in interpreted languages, but it is clear that change is needed.

EXISTING SOLUTIONS

As the number of cores on modern processors grows ever higher, so do the number of available SIMD units. Most native compilers allow for explicit declaration of SIMD instructions in the form of intrinsics, while others go even further by extending the compiler front end with vector pragmas.⁵ This closes the gap between the application and the underlying architecture. One study has shown it is possible to create a common set of macros for calling many different SIMD units, including MMX, SSE, AltiVec, and TriMedia.¹³ This set of macros, called MMM, provides an abstraction for SIMD unit programming that makes programs more portable across hardware platforms. Although this initial approach did provide code portability, it did not address interpreted languages. A similar approach is also available at the compiler level.¹⁰ Whereas some compilers can selectively specialize code at compile time using late binding,³ it is simpler to let the user make parallelism explicit within the code, resolving the parallelism at runtime.

Interpreted languages do not expose vector functionality to the programmer in a transparent way. This is becoming an important problem, since high-performance computations are increasingly being carried out using languages such as Java rather than Fortran.² One solution addressing this performance gap for interpreted languages is AMD's Aparapi,¹ which provides thread-level parallelism and access to video-card acceleration but no SIMD support. Intel has also exposed some native SIMD support in its recent Intel Math Kernel Library in the form of a library package.⁶

There is an argument for writing your own custom native interface for SIMD access rather than using a standardized API or virtual machine extensions. One of the simplest solutions out there today is to find hotspots in Java code using runtime profiles and add JNI (Java Native Interface) calls to the offending code, effectively offloading the inefficient operation to faster native code. Unfortunately, this approach works only when the processor architecture is known to the programmer, and it adds compatibility problems when upgrading to a new architecture.

ARGUMENTS FOR CHANGE

The arguments for supporting the inclusion of premapped vector intrinsics within interpreted

languages are faster application runtime, lower cost, smaller code size, fewer coding errors, and a more transparent programming experience.

Hardware continues to change, but interpreted languages are not keeping pace with these changes, often relying solely on the default data path. The integration of SIMD instructions into virtual machines and their matching language specifications allows for a larger utilization of available resources. Direct vector operation mappings will yield higher utilization of the SIMD unit, and therefore the code will run faster. Multiprocessors today and in the future will contain a SIMD hardware core for each processor, magnifying the disparity between sequential and parallel code. For example, in Intel's Core i7-920, an x86-64 processor with four physical cores, each core contains an SSE 4.2 instruction-set extension. A virtual machine can take advantage of four SSE units at once, outperforming the four single-processor cores on throughput.

As cloud-computing providers such as Amazon and Google have begun to take hold, resource utilization has become a key issue. These organizations do not know ahead of time the potential client workloads; therefore, resource utilization improvement by programming languages will offer great benefit. As the interpreted languages running on these clusters improve, the need to expand the cloud more readily is reduced, with a corresponding cost savings.

As the user gains more control over vectorized data types, the coding experience when exploiting SIMD operations becomes more transparent. The code size, measured in lines of code, will be reduced because SIMD operations on vectors can now be performed inside the virtual machine instead of inside every program. This is the same reason that stacks, queues, arrays, hash maps, strings, and other data structures reduce code size. These classes are encapsulating and abstracting complexity. This encapsulation and abstraction also means fewer errors in coding. Whereas the state of the art requires the user to make a native interface call and write C/C++ code to access the SIMD instructions explicitly, a standardized API and the proposed virtual machine integration avoid complex debugging and instead expose SIMD intrinsics.

ADDRESSING SIMD INCLUSION CONCERNS

Platform compatibility can be achieved by adding native SIMD mappings to the virtual machines of target platforms. There should always exist a default mapping where SIMD operations are mapped at runtime to sequential code if the appropriate hardware is not available.

Virtual machines such as the HotSpot JVM (Java virtual machine) from Oracle already contain optimizations to identify SIMD opportunities on the fly. The JIT compiler must run at a near realtime pace and, therefore, cannot spend a lot of time finding an optimal SIMD mapping at runtime. Simple SIMD operations written in Java are easily picked up by the JVM, but more complex vector manipulation sequences of operations can be easily overlooked.¹¹ Many complex applications are not transformed well into parallel SIMD code by optimizing compilers. We argue that the developer often knows which operations can be executed in parallel and should be able to code them in a standardized fashion that will utilize the available SIMD functionality.

For example, consider a parallel add instruction that explicitly uses the SIMD unit to add two arrays of known data type:

```
jSIMD_API.add(foreground_pixel_blue,background_pixel_blue,foreground_pixel_blue);
```

The user knows that the source and destination of the operation are arrays of the same type and length, and this type of operation is most beneficial when sequences of SIMD operations can be chained together, avoiding transfers between the SIMD and main CPU register files.

Another source of resistance that we have heard from industry and research personnel concerns the possibility of using a dedicated graphics card for GPGPU (general-purpose computing on graphics processing units). Examples of this technology are Nvidia's CUDA and ATI's STREAM. In the context of Flash, Adobe recently announced that its ActionScript can now use Nvidia graphics cards.⁸ This alternative approach is legitimate for those users who have such a graphics card supporting the required features. Unfortunately, this means the user has to purchase an external video card that is often expensive and requires additional power. In the cloud-computing context this can add up to a high number of video cards with associated costs. Also note that each transaction requires system bus access, whereas the SIMD unit is right on the processor die and ships with nearly every processor. Future generations of processors may include GPUs on the die, but until that is the case for existing infrastructures, SIMD is a low-hanging fruit, not fully utilized for getting more computations per core. Most significant is the fact that GPUs capable of such a feat are not as widely distributed as processors. Intel commands a 50-percent market share with its integrated graphics products.¹² As multicore systems become more prominent, the possibility of using multiple SIMD units becomes even clearer.

The arguments against including SIMD intrinsics in interpreted languages can be easily overcome with standardized parallel virtual-machine extensions and specifications. Hopefully, these ideas will become readily available in future language evolutions.

USING SIMD IN JAVA

In response to the need for SIMD in interpreted languages, we designed an API called jSIMD for mapping Java code to SIMD instructions using vectorized data of various data types. JNI is a feature provided within Java to allow access to native code. It is used in the back end as a bridge between the Java-familiar API seen by the programmer and the SIMD mappings compiled as native code. Using an image-processing program as an example, we observed a 34-percent speedup over traditional Java code. Earlier tests on simpler and purely mathematical constructs have yielded speedups of two to three times.¹¹

An overview of the API is shown in figure 1. Once a transaction of operations on vectors is built up, the user code tells the API to initiate the desired operations. The API identifies the available operating system and SIMD units in order to decide when to execute API calls in Java and when to pass the calls to the dynamic libraries with SIMD mappings for parallel execution. If parallel execution is possible, the API makes JNI calls to dynamic libraries, which carry out SIMD instructions on data in the Java memory space. The SIMD native library can be recompiled for a different target architecture through gcc or by using a prepackaged binary. Generic source code has been used to facilitate simple cross-compilation.

MOTIVATING EXAMPLE

Consider a motivating example that uses the jSIMD API to obtain a speedup over an out-of-the-box Java solution. Alpha blending, used for creating transitions in video processing or blending two individual images, is one example of an algorithm that can be moderately accelerated through the use of SIMD instructions. There are many such parallel data-processing applications that are easy

FIGURE 1

High-level Overview of the jSIMD API

to write using the SIMD paradigm. Examples of SIMD tasks that are inherently parallel include 3D graphics, realtime physics, video transcoding, encryption, and scientific applications. Selected versions of these applications are usually supported by custom native code in the virtual machine, whereas our solution gives the programmer the ability to express any algorithm, not just the ones built into the interpreter.

We obtained execution profiles using Intel's VTune Performance Analyzer,⁷ which can be used to profile and analyze various executable applications. We used it to observe the number and types of SSE calls performed by the JVM alone. An alpha-blending program was executed using several standard-size images (640 x 480 - 1920 x 1080 pixels); 1,000 samples for each test executed on an Intel Core 2 Duo E6600 with 2-GB DDR2 RAM running Windows XP Pro SP3. Using jSIMD resulted in an average speedup of 34 percent, and a large number of SSE calls as expected. Also, no SIMD instructions were executed when using the out-of-the-box Java solution, while the results when using the jSIMD API showed that the number of retired SIMD instructions was in the millions and saved several milliseconds per frame. For video transcoding this is a significant performance improvement. The linear relationship between retired SIMD instructions and pixel count means that the API works well at large and small scales.

These observations show that exposing SIMD intrinsics will improve execution time by calling more SIMD instructions. The current jSIMD implementation yielded a speedup below the

anticipated level, based upon a maximum of four concurrent operations within SSE for the data types and processor that we used. The speedup is still significant, considering that no changes to the underlying system architecture were needed and that the changes to the user code were relatively simple and natural. As it is impossible to guarantee that arrays remain pinned in the JVM⁹ because of the garbage collector, memory copies occur occasionally, as confirmed through analysis.

MAKING IT HAPPEN

Some of the problems that arose during the development of the jSIMD API were dependencies between SIMD code and regular Java code, and multiple instantiations of the API. The integrity of the vector registers during program execution is another area of concern. We found that even though Java does make SIMD calls on its own, the JVM will not interrupt the JNI call to our API, and therefore it will not replace any of the contents of the SSE registers on the fly.

The use of SIMD registers can be inefficient unless data transfer between memory and the SIMD unit is reduced. Looking at lists of SIMD operations as transactions allows for further analysis, weighing the performance gain versus the overhead cost. One drawback to our approach is that interlacing SSE calls with regular Java code may cause thrashing of register files. Our current solution requires the programmer to write all SSE code in one continuous block so that the JVM does not need to execute while the JNI call is performed.

When calling the API to perform a sequence of SIMD operations, the API packages the operations into a transaction using a simple sequential list scheduling algorithm, and then passes off all of the instructions and data by reference to the C program, which executes the SIMD instructions. Dependencies with regular Java code, such as casting before an API execute statement, must occur outside of a transaction unless they are done using the API. Dependency and anti-dependency resolutions will further improve execution time and utilization.

LOOKING TO THE FUTURE WITH SIMD

Interpreted languages can expose vector functionality to the programmer, and the results will be faster, smaller, and simpler code as demonstrated by a practical application of this approach using Java. Furthermore, better SIMD utilization within cloud-computing infrastructures has the potential to reduce costs significantly. Improving the scheduling algorithm within individual transactions is a future direction that will indeed increase performance and throughput. Another clear next step is to take advantage of multiple cores at the same time in a real cloud-computing infrastructure.

Our results can be generalized and included in many virtual machines. For example, Flash would clearly benefit from further manual SIMD intervention by the developer for ActionScript computational segments. PHP and JavaScript can also derive benefits from such an approach in order to increase the speed of Web applications. More generally, if you create a virtual machine, you should allow explicit access to generic SIMD instructions. Since you have paid for the SIMD unit inside your server, you might as well let your programmers use it. Although this work is still in progress, we are confident that it will be widely adopted for interpreted languages.

Users can easily identify parallel operations on vectors and arrays. Interpreted languages need not be forcefully architecturally agnostic and opaque. The time has come for virtual machines to embrace their underlying architectures so that data centers and high-performance applications can fully utilize the available processing infrastructure. It is reasonable to expect existing virtual-

machine instruction sets to include generic SIMD operations. Returning some control to the programmer, especially for inherently parallel vector operations, is an easy step toward a transparent programming experience. This argument should not be confused with the age-old assembly-versus-C question. Generic SIMD mappings can retain the abstraction that already exists in interpreted languages; all the while the user is unaware of the exact hardware mapping that is taking place. Users are simply given more control to specify what they believe to be parallel code, negating the required rediscovery used by real-time algorithmic approaches.

SIMD units available in nearly every desktop and server processor are underutilized, especially when using interpreted languages. Giving more control to the programmer and the programming syntax allows for successful and simple mappings with performance increases. Virtual machines must address the current challenges of integrating SIMD support. **Q**

REFERENCES

1. AMD. Aparapi; <http://developer.amd.com/zones/java/aparapi/>.
2. Amedro, B., Bodnartchouk, V., Caromel, D., Delbé, C., Huet, F., Taboada, G. L. 2008. Current state of Java for HPC. Sophia Antipolis Cedex, France; <http://hal.inria.fr/docs/00/31/20/39/PDF/RT-0353.pdf>.
3. Catanzaro, B., Kamil, S. A., Lee, Y., Asanović, K., Demmel, J., Keutzer, K., Shalf, J., Yelick, K. A., Fox, A. 2010. SEJITS: getting productivity and performance with selective embedded JIT specialization. Technical Report UCB/EECS-2010-23. EECS Department, University of California, Berkeley; <http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-23.html>.
4. Cheema, M. O., Hammami, O. 2006. Application-specific SIMD synthesis for reconfigurable architectures. *Microprocessors and Microsystems* 30 (6): 398 – 412.
5. Codeplay. VectorC Compiler Engine; <http://www.codeplay.com>.
6. Intel Software Network. 2010. Intel AVX optimization in Intel MKL V10.3; <http://software.intel.com/en-us/articles/intel-avx-optimization-in-intel-mkl-v103/>.
7. Intel Software Network. 2010. Intel VTune Amplifier XE; <http://software.intel.com/en-us/intel-vtune/>.
8. Nvidia. 2009. Adobe and Nvidia announce GPU acceleration for Flash player; <http://www.adobe.com/aboutadobe/pressroom/pressreleases/200906/060209AdobeandNvidia.html>.
9. Oracle. 2010. JNI enhancements introduced in version 1.2 of the Java 2 SDK; <http://download.oracle.com/javase/1.3/docs/guide/jni/jni-12.html#GetPrimitiveArrayCritical>.
10. Orc (Oil Runtime Compiler); <http://code.entropywave.com/projects/orc/>.
11. Parri, J., Desmarais, J., Shapiro, D., Bolic, M., Groza, V. 2010. Design of a custom vector operation API exploiting SIMD within Java. Canadian Conference on Electrical and Computer Engineering (May).
12. Ranganathan, L. 2009. 3D gaming on Intel Integrated Graphics; <http://software.intel.com/en-us/articles/3d-gaming-on-intel-integrated-graphics/>.
13. Rojas, J. C. 2003. Multimedia macros for portable optimized programs. Ph.D. dissertation, Northeastern University.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

JONATHAN PARRI (jparri@uottawa.ca) is a Ph.D. candidate at the University of Ottawa and a senior member of the Computer Architecture Research Group. His current research focuses on design space exploration in the hardware/software domain, targeting embedded and traditional systems. He has been employed in many capacities from developer and technical writer to research engineer.

DANIEL SHAPIRO (dshap092@uottawa.ca) is a Ph.D. candidate at the University of Ottawa and a senior member of the Computer Architecture Research Group. His research interests include custom processor design, instruction-set extensions, IT security, and biomedical engineering. He has extensive industry experience in realtime system design with complex algorithm integration.

MIODRAG BOLIC (mbolic@site.uottawa.ca) is an associate professor at the School of Information Technology and Engineering, University of Ottawa. His research interests include computer architectures, radio frequency identification, and biomedical signal processing. He has published more than 80 journal and conference papers and edited one book. He is director of the Computer Architecture Research Group at the University of Ottawa.

VOICU GROZA (groza@site.uottawa.ca) joined the University of Ottawa in 1997, where he is currently with the School of Information Technology and Engineering. He is the author and coauthor of more than 200 technical papers. His research interests include hardware/software co-design, biomedical instrumentation and measurement, along with reconfigurable computing. He is the codirector of the Computer Architecture Research Group at the University of Ottawa.

© 2011 ACM 1542-7730/11/0200 \$10.00