Usage of SIMD Processor Extensions

Houffaneh Osman
halio029@uottawa.ca
School of Information Technology and Engineering
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5, Canada

***Abstract*—Computer architectures are classified depending on their implementation. This parper describes SIMD architecture and its implementation in computer systems when dealing with performance time. The[illegible]a multitude of tools that one can use in order to automate SIMD implementation such as compilers. In multi-processing and multi-core, utilizing SIMD instruction when implementing a program with heavy computation is critical for better performance. Computer systems are able to utilize SIMD extensions when operating systems deem it to be necessary for which computations are performed on multiple processors.**

I. Introduction

A computer architecture can be classified into four main categories. These categories are defined under the Flynn's Taxonomy [12]. A computer architecture is classified by the number of instructions that are running in parallel and how its data is managed. The four categories that a computer architecture can be classified under are.:

1) Single Instruction, Single Data (SISD)
2) Single Instruction, Multiple Data (SIMD)
3) Multiple Instruction, Single Data (MISD)
4) Multiple Instruction, Multiple Data (MIMD)

II. Prior Art

The Single Instruction data stream, Single Data data stream (SISD) is an architecture that computes for only one data stream at an instance. This can be related to a looping function for which accepts an array and performs computation with no parallelization, in other words, an implementation of a sequential program. The only issue for this architecture is how data is managed, such as the amount that it can handle at one time and its computational time. When using this computer architecture in a heavy data computation environment, there's a drawback on the speed of the performance time.

III. SIMD processors

This architecture allows the possibility for faster computation by taking a data set as a whole, creating multiple N subset of it, and distributing it to multiple N numbers of processors. The output of the computation will be equivalent to performing computation with one processor and the data as a whole, but this imposes a drawback when it comes to performance time. Of course this depends on the availability of N numbers of processors, but in recent trends shows that most or a high percentage of computer system are provided with multiple or more than one processor. The general ideology can be seen on Fig.1.

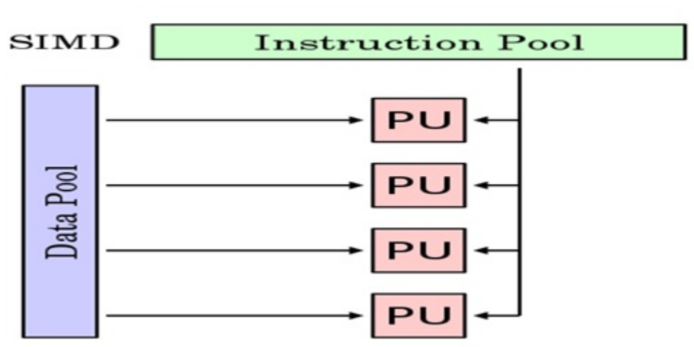

Fig. 1. General SIMD architecture [18]

A. Implementing a SIMD architecture

Two types of SIMD architectures exist: true SIMD and pipelined SIMD. True SIMD architectures can be determined by its usage of distributed memory or shared memory. Both true SIMD architectures possess similar implementation as seen in Fig.2, but differ on placement of processor and memory modules.

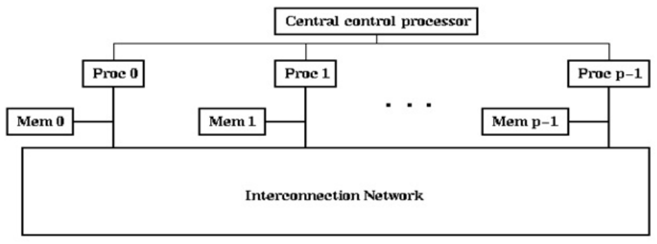

Fig. 2. SIMD architecture [11]

A true SIMD architecture with distributed memory possesses a control unit that interacts with every processing element on the architecture. Each processor possesses their own local memory as observed in Fig.3. The processor elements are used as an arithmetic unit where the instructions are provided by the controlling unit. In order for one processing element to communicate with another memory on the same architecture, such as for information fetching, it will have to acquire it through the controlling unit. This controlling unit handles the transferring of the information from one processing element to another. The main drawback is with the performance time where the controlling unit has to handle the data transfer. However, [illegible] simpler to add a processing element and memory as an entity.

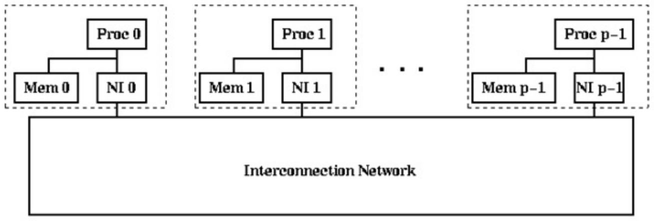

Fig. 3. True SIMD architecture: Distributed memory [11]

Another true SIMD architecture is the shared memory architecture. This architecture is similar to the distributed memory as shown previously. In this architecture, a processing element does not have a local memory but instead its connected to a network where it can communicate with a memory component. Fig.4 shows all the processing elements connected to the same network which allows them to share their memory content with others. In this architecture, the controlling unit is ignored when it comes to processing elements sharing information. It still, however, provides an instruction to the processing elements for computational reasons. The disadvantage in this architecture is that if theres a need to expand this architecture, each module (processing elements and memory) has to be added separately and configured. However, this architecture is still beneficial since it improved performance time and the information can be transferred more freely without the controlling unit.

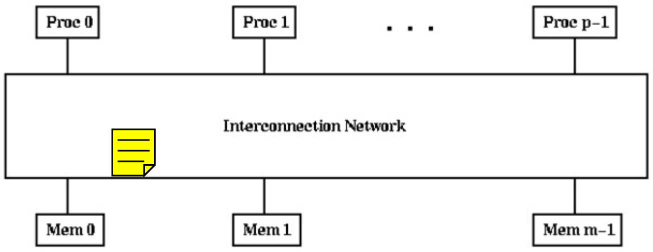

Fig. 4. True SIMD architecture:Shared memory [11]

The other SIMD architecture is the pipelining architecture. This architecture implements the logic behind pipelining an instruction as observe on Fig.5. Each processing element will receive an instruction from the controlling unit, using a shared memory, and will perform computation at multiple stages. The controlling unit provides the parallel processing elements with instructions. The sequential processing element is used to handle other instructions.

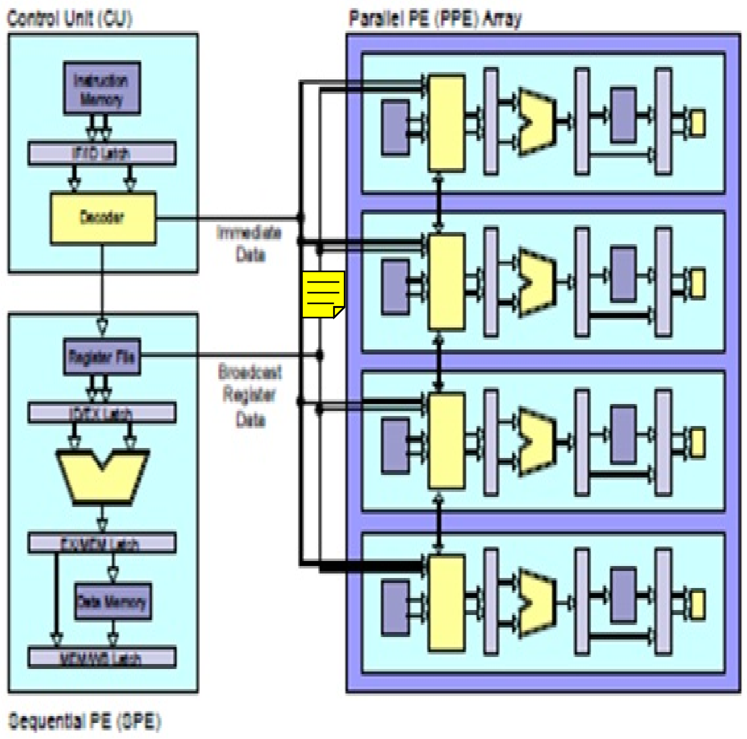

Fig. 5. Pipelined SIMD architecture [17]

B. Programming the SIMD architecture

From the overview of the previous section, a deduction leads to [illegible]t this architecture can be implemented using parallel programming. Cell Broadband Engine (Cell BE) processor is an architecture developed by Sony, Toshiba, and IBM. This architecture was developed for usage with a product demands heavy processing such as gaming consoles or higher definition television and image processing devices.

The Cell BE is a single chip multi-core architecture with nine processors using a shared coherent memory. The nine processor of the Cell BE are configured to be a PowerPC Processor Element (PPE) or a Synergistic Processor Element (SPE). For this architecture, [illegible]ere 1 PPE and 8 S[illegible] The PowerPC Element is 64-bit and capable of running an operating system and applications. While being utilized as a 64-bit architecture, it can also be configured to run 32-bit systems. The synergistic processsing element is used to run heavier computing applications. Each processor configured as a SPE is runni[illegible]dependent application, and all have access to the shared memory. The premise of SIMD is that it utilizes data-level parallelism on multiple processing elements. The PPE supports Vector Media eXtension (VMX) instruction set, and the SPE is implemented as a SIMD architecture. Communication between the PPE and SPE is performed using the mailbox concept for message passing.

SIMD programming instructions on the Cell BE are programmed with the C/C++ language. Vector programming instructions can be used with the Cell BE architecture. For a SIMD operation to be deemed processable, the computation being done on the data must be with the same instruction. The difference between a scalar operation and a SIMD operation can be observed on Fig.6. On a scalar operation, the four addition instrution will be executed individually while for a SIMD operation only one instruction is required for all four data bits.

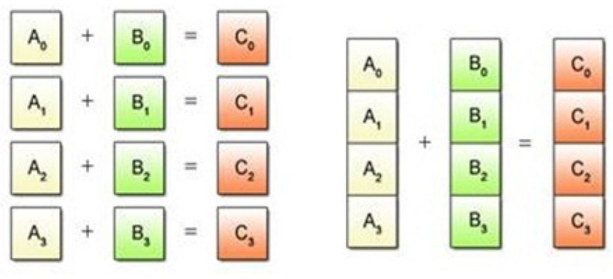

Fig. 6. Vector addition : Scalar vs SIMD implementation [3]

For vector programming, the data being computed will be in vector form instead of a regular array. The Cell BE is using 128-bit fixed-length vectors for the following data type:

1) Signed/Unsigned char
2) Signed/Unsigned short
3) Signed/Unsigned int
4) Signed/Unsigned long
5) Float
6) Double

For example, following is a vector addition using scalar and SIMD programming.

```
int a[4] = { 1, 3, 5, 7 };
int b[4] = { 2, 4, 6, 8 };
int c[4];

c[0] = a[0] + b[0];      // 1 + 2
c[1] = a[1] + b[1];      // 3 + 4
c[2] = a[2] + b[2];      // 5 + 6
c[3] = c[3] + c[3];      // 7 + 8
```

Fig. 7. Vector addition in scalar implementation [3]

On Fig.7, the scalar program shows that a programmer has to manually implement the addition for each value of the arrays. This represents a drawback on the number of lines of code to write and the software performance time. Implementing a for-loop that will parse through the array and perform the addition will reduce the amount of line code that a software developer will write but not the performance time. On Fig.8, the SIMD program will compute using the *vec_add* function that will add the data as one whole sum of data with another whole sum of data.

```
int a[4] __attribute__((aligned(16))) = {1,3,5,7};

int b[4] __attribute__((aligned(16))) = {2,4,6,8};

int c[4] __attribute__((aligned(16)));

__vector signed int *va = (__vector signed int *) a;

__vector signed int *vb = (__vector signed int *) b;

__vector signed int *vc = (__vector signed int *) c;

 // 1 + 2, 3 + 4, 5 + 6, 7 + 8
*vc = vec_add(*va, *vb);  \\
```

Fig. 8. Vector addition with SIMD implementation [3]

The SIMD operations can compute greater data than the scalar operations. Furthermore, if one wants to expand the data needed to be computed, its only a procedure to add more data and not changing any portion of code. This is not true for scalar operations. As shown in Fig.9, the addition function will compute, in parallel, each register content and stored them in the resulting vector.

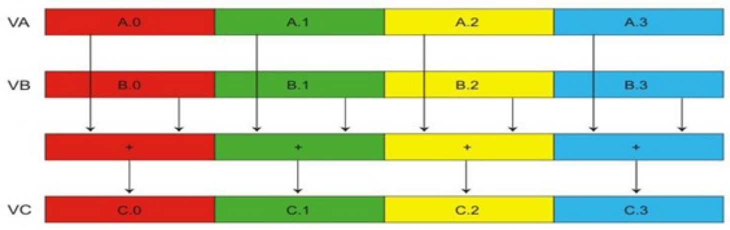

Fig. 9. Vector Programming : *vec_add* instruction [9]

Some VMX instruction that would replace many scalar instructions is as follows and not limited to:

1) Arithmetic : vec_sub (a,b) *Substract two elements*
2) Logic : vec_and (a,b) *Perform AND operation*
3) Compare Operation : vec_cmpeq(a, b) *Compare vector a with vector b for equality*

C. Recognition of possible SIMD Instructions

When a developer is writing a software where theres a need to use parallelization, it must be manually implemented into the software. This would include running a portion of the code on multi-core and making use of parallel threads. This can be a drawback since a developer is constantly checking what can be parallel and what cannot and manually implementing it.

Theres existence of tools that would recognize when a portion of a program can be run using SIMD concept in order to speed up the performance time. Some C/C++ compilers support automatic SIMD recognition. These tools also convert a sequential of the program to parallel processing. Two concepts exist for compilers for SIMD recognition in a program: Auto-parallelization and Auto-vectorization.

Auto-parallelization helps utilize multiple thread implementations when using a sequential program. The multiple threads created will be running on multiple processors. Certain compilers are provided with auto-parallelization support. The Intel C++ Compiler, with specification from the user, will take a sequential program and analyze it using auto-parallelization profiling. When a user is developing a program with this compiler, the user can specify the number of cores available to the compiler. The Intel C++ compiler also supports OpenMP (Open Multi-Processing) development. The PGI Cluster Development Kit (PGI CDK) offers supports for OpenMP and auto-parallelization for AMD Opteron and Intel Core 2 multi-core processor. This PGI CDK is, however, a commercial product from The Portland Group.

Auto-vectorization is capable of detecting a low-level operation, such as a for-loop, and processes it to use multiple processes. For example, the compiler will convert a sequential operation to be computed using 2 to 16 elements in parallel. The GNU Compiler for C/C++ have integrated a support for auto-vectorization. If one was developing a software on the PowerPC with a portion of the code being vectorizable loops, the compiler will provide information on what can and cannot be vectorizable. Analyzing the Fig.10, the compiler will provide the information marked as ”feature” to the programmer to inform of the possibility of auto-vectorization usage. The developer can include auto-vectorization call with the loop. This is called loop vectorizer as per the GCC compiler.

```
int a[256], b[256], c[256];
foo (int n, int x) {
   int i;

   /* feature: support for
    unknown loop bound  */
   /* feature: support for
   loop invariants  */
   for (i=0; i<n; i++)
      b[i] = x;
   }

   /* feature: general loop
   exit condition  */
   /* feature: support for
   bitwise operations  */
   while (n--){
      a[i] = b[i]&c[i]; i++;
   }
}
```

Fig. 10. GCC support for auto-vectorization [6]

D. Using SIMD extensions

The INTEL SSE is a commonly used SIMD instruction which stands for Streaming SIMD Extension. It is a SIMD extension to the x86 architecture for which added 128-bit registers and 70 instructions. This extension allows one to utilize it where heavier computation is required. For Digital Signal Processing (DSP), were able to compute on larger digitized samples without having to worry about the draw-back of quality. Another usage is for security and corruption protection where usability of the SSE instruction can be found with CRC (Cyclic Redundancy Check), MD5 (Message Digest Algorithm), SHA (Secure Hash Algorithm). In image processing, usage of the SSE instruction is necessary for processing frame rates at a higher speed. When working with pixel, a consideration can be taken on the resolution and the pixel encoding in order to provide, with a certain algorithm, desired image.

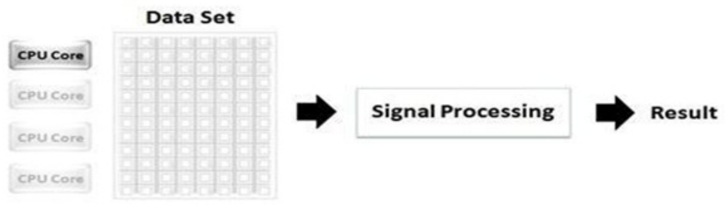

Fig. 11. Digital Signal Processing with one core [8]

In data parallelism, the amount of data and ho[illegible]s computed is important. Data parallelism is a way to take a data as whole and creating multiple subsets of it for computation. As shown in Fig.11, if one were to compute the whole data with a specific instruction on one processing element, this will take a huge amount of time. However, comparing this to a programming method where subsets of data is taken and computed with the same instruction but on multiple cores will improve performance time. This would mean sending multiple subset of data, for example, a digitized sample of a signal, to multiple cores, computed them with the same instruction and combining each result output for each subset of data as observe on Fig.12. This method would greatly improve on the processing time if a comparison is made with the previous implementation with only one processing element usage.

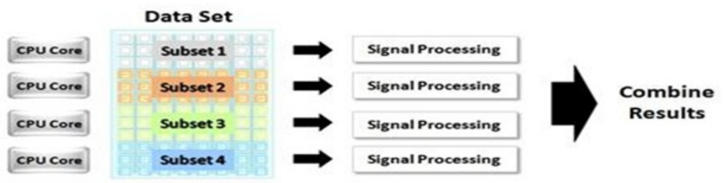

Fig. 12. Digital Signal Processing with multiple cores [8]

E. Operating system SIMD scheduling

Since Intel Pentium III introduction, an operating system comes with supported with the SSE instruction in order to utilize the SIMD concept. Operating systems are capable of scheduling multiple threads for data computation and running application. And the multiple threads can be scheduled to run on separate cores. The multiple threads are simultaneously executing the same instruction on shared execution resources.

IV. Conclusion

SIMD instructions are highly used for field such as digital signal processing or in gaming consoles. They allow for more faster and multiple computation in this field where sacrifice cannot be made on the delay of time. Many compilers support SIMD instructions and others also allow a developer to nest a instruction call inside a portion of a program for heavier computation in order to request to the compiler for usage of the SIMD extension.

References

[1] Intel Press, ”Multi-Core Programming : Increasing Performance through Software Multi-threading,” pp. 2–6 – 11–13, Apr 2006.

[2] Intel Corp. ”Intel C++ Compiler 8.1 for Linux, Internet: ftp://download.intel.com/support/performancetools/c/linux/sb/clin81_relnotes.pdf, 2004 pg 1–9.[2010-10-24]

[3] Linux Kernel Organization, Cell Programming Primer : Basics of SIMD programming,”, *Documents of PS3 Linux Distributor's Starter Kit*. Internet: http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingTutorial/BasicsOfSIMDProgramming.html, 2006,2007,2008 [Oct. 24, 2010].

[4] C. Chen, R. Raghavan, J. Dale, E. Iwata, Cell Broadband Engine Architecture and its first implementation,”. Internet: http://www.ibm.com/developerworks/power/library/pa-cellperf/, Oct. 2005 [Oct. 24, 2010].

[5] H. Chang, C. Cho, S. Wonyong, “Performance Evaluation of an SIMD Architecture with a Multi-bank Vector Memory Unit,”, *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, oct. 2006, pp. 1520-6130.

[6] GCC GNU Project, Auto-vectorization in GCC,”. Internet: http://gcc.gnu.org/projects/tree-ssa/vectorization.html, Aug. 2010 [Oct. 24, 2010].

[7] Intel Software Network, Performance Tools for Software Developers - Auto parallelization and /Qpar-threshold,”. Internet: http://software.intel.com/en-us/articles/performance-tools-for-software-developers-auto-parallelization-and-qpar-threshold/, Jul. 2009 [Oct. 24, 2010].

[8] National Instruments, Programming Strategies for Multicore Processing: Data Parallelism,”. Internet: http://zone.ni.com/devzone/cda/tut/p/id/6421, Nov. 2008 [Oct. 24, 2010].

[9] A. Lanterman, Multicore and GPU Programming for Video Games: Developing Code for Cell - SIMD”. Internet: http://users.ece.gatech.edu/ lanterma/mpg09/, Fall 2010 [Oct. 24, 2010].

[10] IBM Corp and Sony Computer Entertainment (2006,2007). ”Software Development Kit for Multi-core Acceleration (Version 3.1). [On-line], Internet: http://public.dhe.ibm.com/software/dw/cell/CBE_Programming_Tutorial_v3.1.pdf, DEC. 2009 [Oct. 24, 2010].

[11] J. Demmel, ”A closer look at parallel architectures: Lecture 9,” Internet: http://www.eecs.berkeley.edu/ demmel/cs267/lecture09/lecture09.html, Feb. 1996 [Oct. 24, 2010].

[12] R. Michael Hord, ”Parallel supercomputing in SIMD architectures,” Boca Raton, FL: CRC Press, c1990

[13] S. . Morse, ”Practical parallel computing ,” Boston : AP Professional, c1994

[14] C. . Leopold, ”Parallel and distributed computing : a survey of models, paradigms and approaches ,” New York : Wiley, 2001

[15] L. Dong-hwan, S. Wonyong, “Importance of SIMD computation reconsidered,”, *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, apr. 2003, pp. 8.

[16] W.C. Meilander,J.W. Baker, M. Jin, “Performance Evaluation of an SIMD Architecture with a Multi-bank Vector Memory Unit,”, *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, oct. 2006, pp. 1520-6130.

[17] H. Wang. ”Design and Implementation of an FPGA-Based Scalable Pipelined Associative SIMD Processor Array with Specialized Variations for Sequence Comparison and MSIMD Operation,”, [On-line],”, Internet: ”http://etd.ohiolink.edu/send-pdf.cgi/Wang%20Hong.pdf?kent1164126183”, Nov. 2006 [Oct. 24, 2010].

[18] Wikimedia Commons. ”SIMD [On-line],”, Internet: ”http://en.wikipedia.org/wiki/File:SIMD.svg,”, Jun. 2007 [Oct. 24, 2010].