Teaching SIMD Instructions using Intel Intrinsics in Computer Organization Course

Satish Puri
Department of Computer Science
Marquette University
Milwaukee, WI
Email:satish.puri@marquette.edu

***Abstract*—This talk focuses on integrating high performance computing topics in Computer Organization class at Marquette University. SIMD parallelization was covered in the class using CPU and GPU. On Intel CPU, Advanced Vector Extension (AVX) was used for teaching vectorization at the assembly programming level. The content was covered in two fifty minutes lectures, one lab, and one homework. On GPU, CUDA was used with image manipulation example. We describe the topics covered along with programming exercises and homework assignments. The course materials, lecture slides and lecture video has been shared online.**

I. Introduction

This lightning talk is based on experience of teaching Computer Organization and Design class at Marquette University for second year undergraduate students in Computer Science. The class follows the "Computer Organization and Design" book written by Patterson and Hennessey [2]. Topics covered are ARM assembly programming, logic operations, integer and floating point representation, memory hierarchy, multi-cores and graphics processing units. Vector instructions and Single Instruction Multiple Data (SIMD) are covered after the chapter on ARM instruction set architecture.

First I will mention the context and motivation. Third chapter of the course book is "Arithmetic for Computers". This chapter has content on subword parallelism. The book chapter shows matrix multiplication functions with and without using Intel Intrinsics [1]. In the twelve pages of the book chapter, ARM and x86 SIMD extensions are covered. Using Advanced Vector Extensions (AVX) in x86 instruction set, upto 3.85X speedup is mentioned for single thread compared to normal matrix multiplication in C running on Intel processor. The book did not chose ARM Neon instructions to teach subword parallelism through programming example. This motivated the use of x86 extensions. Rather than directly teaching SIMD parallelism for matrix multiplication, simpler examples were created for teaching.

Motivation: In the book, subword parallelism appears towards the end of the chapter (subsections 3.6 to 3.8). The book chapter does not provide a complete program, only a function is given to illustrate subword parallelism. So, the motivation was to complement the existing material with additional contents like lecture slides, programming labs, quizzes and homework assignments targeted towards teaching subword parallelism in more detail. Additional content on vector data types, SIMD registers, intrinsic functions and memory alignment was developed.

In the class, Single Instruction Multiple Data (SIMD) concept is illustrated by showing scalar and vector assembly pseudo-code to do vector addition where two input arrays are added element-wise to produce output array. Embarrasingly parallel tasks like image manipulation is described to motivate SIMD processing. SIMD functions are used in C code first and then assembly code produced by compiler is shown. First Intel Intrinsics was covered and then towards the end of the course, GPU architecture was covered with a simple CUDA program.

Intrinsics are C-functions to perform data movement, logic and arithmetic computations. Intrinsics get translated to vector instructions by compiler. There are few benefits of starting with Intrinsic functions rather than vector instructions directly. First of all, writing a program using C functions is easier. ARM instruction set was taught in assembly level following the format of the book. So, programming exercises for SIMD instructions based on x86 assembly would have been difficult. Therefore, rather than teaching second x86 assembly language, teaching intrinsics was a simpler approach. To write a full fledged program, it is easier to use C functions instead of vector instructions in assembly. Examples of Advanced Vector Extensions (AVX) were shown in class. Programming assignments and laboratory assignments were based on AVX as well.

The lectures were separated into two distinct parts:

Part 1 covered the following topics:

- Conceptual ideas on subword parallelism and SIMD instructions.
- Few intrinsic functions to load, store and compute.

Part 2 covered the following topics:

- Loop transformation to show how to apply intrinsics in a for loop. An example was shown to demonstrate loop unrolling technique. The example showed how unrolling four iterations of a *for* loop makes it easy to apply intrinsic function for double precision floating point type where width of the register is 256 bits.
- Aligned memory allocation.
- Demonstration on Intel processors.
- Quiz, lab problems and HW assignments.

Table below shows one to one mapping between a data type in C (scalar) and a corresponding data type in AVX (vector) that was covered in class.

C data type	AVX data type
int	__m256i
float	__m256
double	__m256d

Table below shows few AVX intrinsic functions and the corresponding assembly instructions covered in the class.

AVX Intrinsic function	Assembly instruction
_mm256_load_pd	vmovapd
_mm256_store_pd	vmovapd
_mm256_mul_pd	vmulpd
_mm256_broadcast_sd	vbroadcastsd
_mm256_set_pd	initialization

Code Listing 1 shows an example to illustrate the intrinsic functions to do load, store and compute operations on array elements.

Code Listing 1 to illustrate intrinsic functions

```
double *arr = {1.0, 2.0 , 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
// Load first four numbers from array arr
__m256d a0 = _mm256_load_pd(arr);
// Load last four numbers from array arr
__m256d a4 = _mm256_load_pd(arr + 4);
//Perform four pairwise additions
```

```
__m256d sum = _mm256_add_pd( a0, a4);
double arr[4] = {0, 0, 0, 0};
```

```
// Copy the contents of sum to array arr
_mm256_store_pd(arr, sum);
```

GCC compiler was used to compile and generate the assembly code in order to see the vector instructions corresponding to the functions used in C program with intrinsics. *immintrin.h* header file is required for compilation. *-mavx* flag has to be used in compilation command. Running the code is same as an ordinary C program. Students were asked to inspect the assembly code produced by compiler using -O2 and -O3 flags.

For standard C-based matrix multiplication, the code generated by the compiler was shown first and then the code generated using the function with intrinsics was shown to compare the vector instructions in the two versions. The parts of the code where compiler generated sub-optimal vector code was highlighted. Following this exemplar, students repeated this exercise for the C program they had written. Students compared the assembly code generated by regular C program with optimized assembly code generated when intrinsics were used.

The difference in vectorization was explained as follows. In sub-optimal code, compiler generated code had XMM registers with 128 bits width. In the version with intrinsics, compiler generated code had YMM registers with 256 bits width. Wider registers have the benefit of packing more data elements in a single register. For data movement, *vmovsd* was generated in the sub-optimal code instead of *vmovapd*. *s* stands for scalar in *vmovsd*. *p* stands for packed in *vmovapd*. The last character *d* is for double precision. *vmovsd* loads 1 element into XMM register, while *vmovapd* loads 4 elements into YMM register. Similarly, *vmulsd* was generated by compiler in the sub-optimal code instead of *vmulpd*. For matrix multiplication, this results in more than 3X speedup difference between the two versions on an Intel processor.

A. Quiz

Here are few sample questions from a quiz:

Q1. World's fastest ARM processor Fugaku by Fujitsu has a SIMD register which is 2048 bits wide. How many double precision floating point data can be stored in one SIMD register?

Q2. What does the array *arr* contain after the code has been executed as shown in Code Listing 1?

Q3. Let us assume that array *arr* is stored in a block of memory with starting location $(3200)_{10}$ and we are trying to use load function as shown below:

__m256d first = _mm256_load_pd(arr + 1);

Will this line of code work?

__m256d first = _mm256_load_pd(arr + 1);

Question 3 is designed to test the memory alignment issue. Memory alignment issue can lead to program crash. 32 byte alignment requirement is violated because address of (*arr*+1) is 3208 which is not completely divisible by 32.

B. Programming Exercises

The course materials containing programming exercises covered in laboratory, quizzes and homework problems has been shared on GitHub [4]. Lecture video has been shared online on Youtube [3].

Programming lab question was polynomial evaluation. Sequential program was provided and the task was to do SIMD parallelization of the loops by adding intrinsic functions. After finishing the coding part, next task was to benchmark the execution time and calculate GFLOPS and speedup with respect to baseline. Image manipulation example using intrinsics was considered but eventually not used because of higher complexity of the code. However, for GPU, image manipulation using CUDA code was shown in the class.

One common problem that was encountered in labs was misspelling in intrinsics that was creating compilation errors.

REFERENCES

[1] Intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[2] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware/software interface*. Morgan kaufmann, 2016.

[3] Satish Puri. Intel intrinsic functions for simd parallelism (vectorization). https://www.youtube.com/watch?v=KABGmD7BJ28t=2750s.

[4] Satish Puri. Teaching intel intrinsics for simd parallelism. https://github.com/satishphd/Teaching-Intel-Intrinsics-for-SIMD-Parallelism.