Accelerating Pre-stack Kirchhoff Time Migration by using SIMD Vector Instructions

Maicon Melo Alves[1], Lúcia Maria de Assumpção Drummond[1] e Reynam da Cruz Pestana[2]

[1]Institute of Computing - Fluminense Federal University (UFF)
Niterói - RJ - Brazil

[2]Institute of Geosciences - Federal University of Bahia (UFBA)
Salvador - BA - Brazil

{mmelo,lucia}@ic.uff.br, reynam@ufba.br

Abstract. *The Pre-stack Kirchhoff Time Migration (PKTM) is a central process in petroleum exploration. As PKTM is computationally intensive, many works have proposed the use of accelerators like GPU and FPGA to improve its execution time. On the other hand, although many off-the-shelf processors are endowed with a set of SIMD vector instructions, few papers tackle the problem considering vectorization and, all of them, consider that compilers can successfully vectorize the code. In this paper, we show that a hand-written Kirchhoff code by using SIMD vector instructions is more efficient than the auto-vectorized code provided by GCC. Experiments considering both real and synthetic datasets showed that our solution is up to eight times faster than the auto-vectorized one. We believe that the proposed strategy can be used together with the other ones to accelerate seismic migration methods in general without new investments in hardware and software.*

1. Introduction

The first step in oil and gas exploration is a process called seismic data processing flow. This process provides a subsurface image from earth that allows an interpretation of geological structures in the target area in order to detect where oil and gas can be found and recovered [Deschizeaux and Blanc 2007]. This process is very important for petroleum industry since each oil and gas well in exploration areas costs several tens of millions of dollars and the production of high-quality seismic images in a reasonable time can significantly reduce the risk of make mistakes about drilling locations. Moreover, these images are important as they can improve the position of wells in a billion-dollar producing oil field [Rizvandi et al. 2011].

Several steps are made in seismic data processing flow such as deconvolution and velocity analysis, but seismic migration is considered the central step in this entire process. Basically, a seismic migration recovers an image of geological subsurface structures by collapsing diffraction energy at discontinuous points and moving reflection events to their geological locations. There are many classes of seismic migration methods with different advantages and constraints. One of the most popular migration methods in oil

and gas industry is Pre-stack Kirchhoff Time Migration (PKTM) that is based on diffraction summation procedure. PKTM is widely used because of its simplicity, efficiency, feasibility and I/O flexibility [Xu et al. 2014].

However, PKTM is computationally intensive. Even in supercomputers, it could take days or weeks to generate the final migrated image for a seismic dataset [Shi et al. 2011]. In other words, a PKTM execution needs to process terabytes of data and requires Gflop-months of computation prior to being interpreted by experts. Consequently, parallel and distributed computing techniques have been applied in PKTM aiming to reduce its execution time [Rizvandi et al. 2011]. Due to data parallelism characteristics of PKTM algorithm, many works, for example [Xu et al. 2014], [Shi et al. 2011], [Panetta et al. 2012], and [Sun and Shi 2012], have proposed the use of computational accelerators such as Graphical Processing Units (GPU) or Field-programable Gate Array (FPGA) to reduce PKTM execution time. In this strategy, generally the computation is accomplished just by GPUs and FPGAs, and CPU is used only for performing control tasks such as moving data from memory to these devices. Although this approach with GPUs and FPGAs achieves satisfactory results, some works as [Panetta et al. 2012] have shown that a new strategy combining computational power of conventional multicore processors and these accelerators can significantly improve PKTM performance.

In addition to computational power provided by a multicore processor, there is another interesting resource available in many off-the-shelf processors that can increase performance of applications. Many processors have been endowed with functional units of Single Instruction Stream Multiple Data Stream (SIMD) which can process multiple data elements at one time by using special sets of hardware instructions. This capability is also called SIMD vector instructions since the basic data unit handled by this instruction set is a vector. Each processor manufacturer has implemented its own set of instructions to provide SIMD capability. In Intel processors, for instance, the set of instructions that support SIMD vector computing is SSE and AVX, whereas 3DNow and Altivec provides such resource in AMD and PowerPC processors, respectively [Rahmad et al. 2011].

Because there is potential data parallelism in PKTM algorithm, one strategy to increase its performance is applying SIMD vector instruction capability available in off-the-shelf processors. A solution that combines this CPU capability with accelerators can strongly increase PTKM performance. Another advantage in using SIMD vector instructions provided by CPU is an immediate acceleration of PKTM in cases where only a conventional CPU cluster or just individual workstations are available. Thus, an expressive increasing in PKTM performance can be achieved without making additional investments in new and specific hardware. This SIMD vector instructions can be used by turning on the auto-vectorization option provided by many compilers such as GCC (Gnu C Compiler), ICC (Intel C Compiler) or XLC (IBM C Compiler) [Maleki et al. 2011]. When this option is enabled, the compiler searches for code structures that can be vectorized, i.e, inner loops processing vectors without data dependency and control flow. However, the compiler auto-vectorization fails in cases where the code does not match the auto-vectorization characteristics searched for the compiler [Mitra et al. 2013]. Thus, in these cases, the programmer must perform a code analysis in order to identify code pieces suitable for vectorization.

In this paper, we present an accelerated version of PKTM by using SSE (*Streaming*

SIMD Extensions) which is a SIMD vector instruction set available in off-the-shelf Intel processors. We performed an analysis of the PKTM code in order to identify steps suitable for applying vectorization provided by SSE. The performance evaluation accomplished in both real and synthetic datasets showed that our solution is up to nine times faster than the traditional code executed in the same machine. Furthermore, we also compared our implementation with the auto-vectorized code generated by GCC which is a free compiler widely used for academic and commercial purposes. The results showed that the speedup achieved by our solution is up to eight times greater than the one achieved by the auto-vectorized version.

More specifically, this work aims to show that an immediate improvement of PKTM performance can be achieved without new investments in hardware or software. Thus, the proposed strategy can be used by PKTM solutions where CPU is used to perform computation as well as in cases where only clusters of CPU or workstations are available. To the best of our knowledge, this is the first paper where the acceleration of PKTM execution by using SIMD vector instructions is evaluated, showing that PKTM is not successfully auto-vectorized by GCC. Therefore, we believe that the lessons learned in this work can be applied in novel and complex migration methods, as the Least Square Migration (LSM), and also in other related problems.

2. Pre-stack Kirchhoff Time Migration (PKTM)

Seismic data processing flow is responsible for converting raw data acquired from earth subsurface which was collected by a process called seismic data acquisition. In this process, a source and a set of receivers are placed in a target area and the distance between the source and each receiver is called *offset*. The source propagates a wave towards earth subsurface which is refracted and reflected when hits on a subsurface rock layer. This process, known as a *Common Shot Acquisition*, is performed several times during a seismic data acquisition. At each shot, sources and receivers are placed at different positions of target area in order to get redundant information about a same point in subsurface [Yilmaz and Doherty 1987].

At discrete periods of time, receivers placed near surface collect energy reflected by subsurface rock layers during an entire shot time propagation. Data collected by a receiver at a discrete time t is known as a *sample* and represents the amplitude energy reflected by a point in subsurface. Supposing a parallel plan, this point, known as *Common Mid Point* (or just CMP), can be considered as been placed in midway between source and receiver positions. A set of samples collected by a receiver during one shot is named *seismic trace* and represents the energy reflected from a CMP during entire shot time propagation. A set of traces of a target area results in a dataset named *seismic section* which is usually saved in a standard file format [Yilmaz and Doherty 1987].

After some pre-processing, the seismic section is ready to be migrated [Yilmaz and Doherty 1987]. Basically, a migration (i) collapses hyperbolic diffractions, (ii) moves dipping reflectors to their true subsurface positions and (iii) increases spatial resolution. A seismic section needs to be migrated because dipping reflector points in subsurface can produce an image that originally does not represent real geological structures. This happens because a dipping point produces a semicircular wave (in accordance with Huygens' Law) which is collected by receivers as a hyperbola. Thus, a resulting image

can apparently have structures not placed in their real positions and structures that do not exist in reality. Thereby, migration corrects this distortions and generates a more accurate image where apparent location of dipping reflectors are moved to their real locations [Yilmaz and Doherty 1987].

Algorithm 1 PKTM algorithm

```
 1: For all Offsets do
        /* Input Traces Loop */
 2:     For all Input Traces do
            /* Filtering Loop */
 3:         For all Cut-frequencies do
 4:             Filter_Values()
 5:             Filter_Input_Trace()
 6:         End For
            /* Migration Loop */
 7:         For all Output Samples do
 8:             Velocity()
 9:             Aperture_Traces()
                /*Contribution Loop*/
10:             For all Output Traces in Aperture do
11:                 Travel_Time()
12:                 Set_Samples()
13:                 Mig_Operator()
14:                 Define_Filters()
15:                 Interpolate()
16:                 Obliquity_Factor()
17:                 Aperture_Angle_Taper()
18:                 Geometrical_Factor()
19:                 Correct_Amplitude()
20:                 Accumulate_Contribution()
21:             End For
22:         End For
23:     End For
24: End For
```

Pre-stack Kirchhoff Time Migration receives a pre-stack common offset section as input and produces a final image in time coordinate. The basic assumption made by Kirchhoff Migration is that any point in seismic section can be considered as a dipping reflector and, consequently, each of these points is located at the hyperbola apex. That assumption means that each point must receive back the energy scattered when this point was excited by a wave front. In other words, this apex must receive the energy contribution of all samples which compose the hyperbola [Teixeira et al. 2013]. The behavior of this hyperbola can be defined by a two-way travel time equation which determines the points (input samples) that contribute to the hyperbola apex (output sample) [Yilmaz and Doherty 1987].

An scalar version of PKTM algorithm is shown in Algorithm 1. For each offset, PKTM executes the *Input Trace Loop* (lines 2 to 23) where each input trace of current offset is processed. Next, PKTM executes *Filtering Loop* (lines 3 to 6) which is composed by

two phases and allows to get anti-aliased input trace samples. In the first phase, the filter values for the current cut-frequency (line 4) is calculated. Next, each input trace sample is filtered (line 5) by using the previously calculated filter values. After that loop execution, there will be distinct filtered versions of the same input trace, each one related to one cut-frequency. Then, a filtered version of the input trace matching an output trace can be chosen. The number of filtered versions (number of cut-frequencies) can be calculated or be informed as an input parameter. After filtering the input traces, PKTM executes the *Migration Loop* (lines 7 to 22). For each output sample, the velocity for current CMP is read (line 8) and traces which compose *aperture* are determined (line 9). An aperture defines the output traces that will receive contributions from a given input trace.

Next, PKTM executes *Contribution Loop* (lines 10 to 21). In the first step of this loop, the two-way travel time is calculated (line 11) to identify which input sample will contribute to current output sample. However, because this calculated travel time is in continuous domain, a set of samples to interpolate is defined (line 12) to transpose it to a discrete domain. After that, the horizontal slowness of migration operator is calculated (line 13) and its value is used as an input parameter to determine the appropriated filters for anti-aliasing (line 14). The amplitude energy is then calculated from an interpolation between filters and samples (line 15). This interpolating process aims to provide an approximated amplitude value, given the chosen filters and the defined set of samples. So, the (i) obliquity factor, (ii) aperture angle taper, and (iii) geometrical spreading factor are calculated (lines 16, 17, and 18, respectively), allowing to correct the resulted amplitude in the next step (line 19). At last, this corrected amplitude energy is accumulated (line 20) in current output sample. A more detailed description of PKTM can be found in [Yilmaz and Doherty 1987].

3. Accelerating PKTM by using SIMD Vector Instructions

In this section we describe our proposed solution to accelerate PKTM execution time. In subsection 3.1 we show how SSE instructions were used to vectorize one single PKTM step. Next, in subsection 3.2 we present the vectorized version of PKTM, describing what steps could be vectorized, and discuss some issues related to this entire process.

3.1. PKTM Steps Vectorization

This subsection shows how SSE instructions were applied to vectorize one single PKTM step. Describing the vectorization of one single PKTM step is enough to understand our vectorization strategy since the other vectorized steps follow this same approach. As an example, we describe how the two-way travel time calculation step was vectorized.

Consider the function *Travel_Time()* (executed in line 11 of Algorithm 1) described in detail in Algorithm 2. This function receives as input parameters the x coordinate of (i) the output and input traces, (ii) the output sample time, (iv) the offset, and (v) the velocity. As output, the Algorithm 2 returns the travel time related to the given output sample time and output trace. Moreover, all parameters and variables are represented as floating point numbers with single precision (data type *float* in C). In line 2 of Algorithm 2 the distance between the input and output traces is computed. The travel time from source to mid-point is calculated in line 3, while the travel time from mid-point to receiver is calculated in line 4 (*sqrt()* and *pow()* calculates square-root and power, respectively). At last, both previously calculated travel times are summed (line 5) and this

resulting two-way travel time is returned (line 6). Thus, this function calculates the two-way travel time for only one output trace at a time. PKTM executes this function for every output trace in aperture (line 11 of Algorithm 1).

Algorithm 2 Scalar Two-way Travel Time Calculation Function

```
    INPUT: output_trace_x, input_trace_x, sample_time, offset, velocity
    OUTPUT: travel_time
1: Function TRAVEL_TIME()
2:     d ← output_trace_x - input_trace_x
3:     ts ← sqrt(sample_time + pow((d+offset)/velocity,2))
4:     tr ← sqrt(sample_time + pow((offset-d)/velocity,2))
5:     travel_time ← ts + tr
6:     Return travel_time
7: End Function
```

In Algorithm 3 the vectorized version of Algorithm 2, called *Vec_Travel_Time()*, is described. *Vec_Travel_Time()* receives four output traces and calculates, at the same time, their respective four travel time values by using SSE instructions. The input parameters of this function are the same of the original scalar version, except for the parameter which receives the output trace. In the vectorized version, the function receives a *float* vector of four numbers, *out_tr_x[4]*, (x coordinate of four output traces) instead of just one *float* number (x coordinate of one output trace) as performed in the scalar version. The *Vec_Travel_Time()* function returns a *float* vector of four numbers where the travel time for each output trace is stored in each vector position. Moreover, all input and output parameters are represented as floating point numbers with single precision.

The SSE instructions used by *Vec_Travel_Time()* allow to process simultaneously 128 bits of data, i.e., four floating point numbers with single precision (32 bits each one). The *Vec_Travel_Time()* procedure accessed the set of SSE instructions by using special functions called *intrinsics* which are provided by Intel libraries. As these intrinsics functions process only specific SSE data types, all data must be copied to this kind of variable before being processed by intrinsics. A description about the intrinsics used by *Vec_Travel_Time()* can be found in [Intel 2015].

At first, all input parameters are copied to variables of a specific SSE data type (variables with *sse* prefix) in lines 2 to 6. After that, the distances between the four input and output traces are calculated at the same time in line 7 (statement equivalent to line 2 in scalar version). Next, the travel time values from mid-point to receiver of the four output traces (corresponding to line 3 of scalar version) are calculated in lines 8 to 12. The travel time values from mid-point to receiver of the four output traces are calculated in lines 13 to 17 (equivalent to line 4 of scalar version). The two-way travel time of the four output traces is then computed in line 18 (line 5 of scalar version). Next, the four two-way travel time values (stored in a SSE vector) are copied to a *float* point vector of four numbers. At last, this vector containing four travel times is returned. Thus, instead of processing just one output trace as performed by *Travel_Time()*, the *Vec_Travel_Time()* function process four travel time values for four output traces simultaneously.

Algorithm 3 Vectorized Two-way Travel Time Calculation Function

```
    INPUT: output_trace_x[4], input_trace_x, sample_time, offset, velocity
    OUTPUT: travel_time[4]
 1: Function VEC_TRAVEL_TIME
      /* Copying input parameters to SSE variables */
 2:   sse_output_trace_x ← _mm_load_ps(output_trace_x)
 3:   sse_input_trace_x ← _mm_set1_ps(input_trace_x)
 4:   sse_sample_time ← _mm_set1_ps(sample_time)
 5:   sse_offset ← _mm_set1_ps(offset)
 6:   sse_velocity ← _mm_set1_ps(velocity)
     /* Corresponding to line 2 of Algorithm 2*/
 7:   sse_d ← _mm_sub_ps(sse_output_trace_x,sse_input_trace_x)
     /* Corresponding to line 3 of Algorithm 2 */
 8:   sse_aux1 ← _mm_add_ps(sse_d,sse_offset)
 9:   sse_aux2 ← _mm_div_ps(sse_aux1,sse_velocity)
10:   sse_aux3 ← _mm_mul_ps(sse_aux2,sse_aux2)
11:   sse_aux4 ← _mm_add_ps(sse_sample_time,sse_aux3)
12:   sse_ts ← _mm_sqrt_ps(sse_aux4)
     /* Corresponding to line 4 of Algorithm 2 */
13:   sse_aux1 ← _mm_sub_ps(sse_offset,sse_d)
14:   sse_aux2 ← _mm_div_ps(sse_aux1,sse_velocity)
15:   sse_aux3 ← _mm_mul_ps(sse_aux2,sse_aux2)
16:   sse_aux4 ← _mm_add_ps(sse_sample_time,sse_aux3)
17:   sse_tr ← _mm_sqrt_ps(sse_aux4)
     /* Corresponding to line 5 of Algorithm 2 */
18:   sse_travel_time ← _mm_add_ps(sse_ts,sse_tr)
      /* Copying travel times from a SSE variable into a float one */
19:   _mm_store_ps(travel_time,sse_travel_time)
20:   Return travel_time
21: End Function
```

3.2. Vectorized PKTM

In Algorithm 4, the vectorized PKTM version is shown. As can be seen in Algorithm 4, the *Contribution Loop* (originally in lines 10 to 21 of Algorithm 1) was divided in two new loops called *Vectorized Contribution Loop* (lines 10 to 18 of Algorithm 4) and *Scalar Contribution Loop* (lines 19 to 23 of Algorithm 4). All vectorized steps were grouped in the *Vectorized Contribution Loop*, executed by iterations of four-by-four output traces. Steps inside this loop compute four output traces simultaneously and makes available their computed data to the scalar steps by storing the resulting data in vectors, where each vector position corresponds to one output trace.

Steps that could not be vectorized were inserted in *Scalar Contribution Loop*, executed by iterations of one-by-one output trace. These steps could not be vectorized due to intrinsic characteristics or SSE limitations. The step of filtering input traces (*Filter_Iput_Trace()*) could not be vectorized because it uses complex numbers which are not supported by SSE. The interpolation step (*Interpolate()*) was not suitable for vectorization because many distinct math calculations are accomplished in only 8 floating point numbers, i.e, vectorization is more suitable for problems where a same operation is per-

Algorithm 4 Vectorized PKTM Version

```
 1: For all Offsets do
      /* Input Traces Loop */
 2:   For all Input Traces do
        /* Filtering Loop */
 3:     For all Cut-frequencies do
 4:       Vec_Filter_Values()
 5:       Filter_Input_Trace()
 6:     End For
        /* Migration Loop */
 7:     For all Output Samples do
 8:       Velocity()
 9:       Aperture_Traces()
          /*Vectorized Contribution Loop*/
10:       For all Output Traces in Aperture step by 4 do
11:         Vec_Travel_Time()
12:         Vec_Set_Samples()
13:         Vec_Mig_Operator()
14:         Vec_Define_Filters()
15:         Vec_Obliquity_Factor()
16:         Vec_Geometrical_Factor()
17:         Vec_Aperture_Angle_Taper()
18:       End For
          /*Scalar Contribution Loop*/
19:       For all Output Traces in Aperture step by 1 do
20:         Interpolate()
21:         Correct_Amplitude()
22:         Accumulate_Contribution()
23:       End For
24:     End For
25:   End For
26: End For
```

formed on many elements. The steps responsible for correcting the resulting amplitude (*Correct_Amplitude()*) and for accumulating the contribution in the output sample (*Accumulate_Contribution()*) were not vectorized because they depend on data provided by the interpolate step, which was not vectorized.

Our implementation presented a little loss of precision if compared with the traditional PKTM code. In Intel processors, the float point arithmetic is always performed in extended precision of 80 bits, even when processing numbers of 32 bits represented with the original IEEE 754 single precision standard [Bryant et al. 2003]. As SSE adopts the original IEEE 754 representation, there is a little lost of precision if compared with the traditional C code. However, there is no significant difference in the final migrated image since single precision is more than adequate for Kirchhoff migration [Levin et al. 2004].

We also used SSE to vectorize data copying between vectors. The conventional intrinsics used to load and store data (*_mm_load_ps()* and *_mm_store_ps()*, respectively) can be used only when data are aligned in 16-bytes boundaries. In some cases when

data are unaligned, we used the intrinsics *_mm_loadu_ps()* and *_mm_storeu_ps()* that can deal with unaligned data. Remark, however, that accessing unaligned data is slower than accessing aligned data in 16-bytes boundaries. Finally, some steps execute trigonometric functions, such as sine and cosine, not supported by SSE. In order to treat this problem, we employed trigonometric functions available in the SSE math library [Pommier 2007].

4. Experimental Results

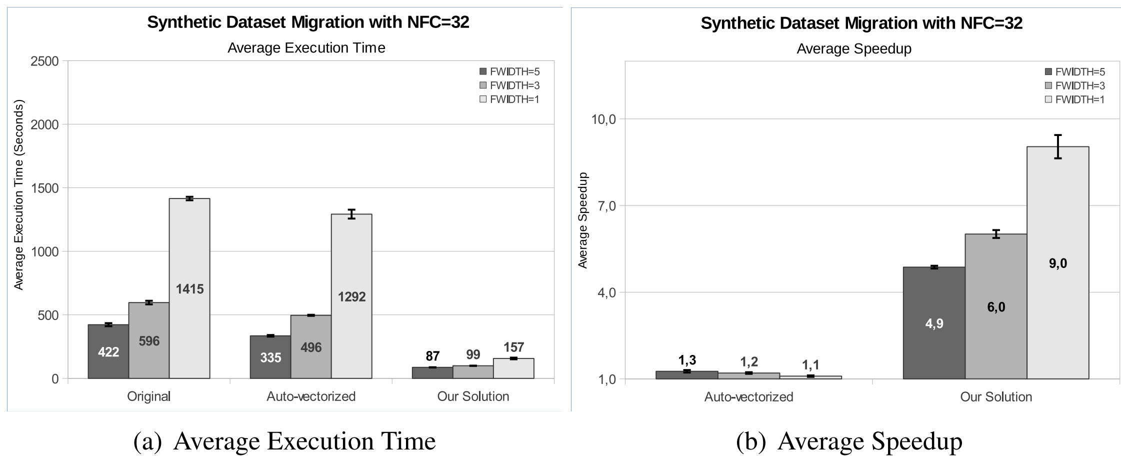
Our PKTM implementation is based on the Seismic Unix, an open source software package for seismic processing [Cohen and Stockwell 2008] that is widely used by petroleum industry and academy. The performance evaluation of our solution was accomplished by executing PKTM over a real and a synthetic datasets. Evaluating different datasets is desired since computational complexity of PKTM is related to their geometry, i.e., different number of offsets, traces per each offset and samples per trace can strongly increase the PKTM execution time. The synthetic dataset and its respective velocity model was created by using a synthetic seismograph, whereas the real dataset is composed of data collected from a real seismic data acquisition accomplished in Brazil [1]. At last, both our solution code and synthetic dataset can be obtained upon requesting to the authors.

The number of offsets, samples per trace, and traces per offset are, respectively, 630, 512, 128 for the synthetic dataset, and 200, 1001, 978 for the real one. Thus, the real dataset has more traces per offset and samples per trace than the synthetic dataset. Thereby, the execution of both *Input Traces Loop* and *Migration Loop* are more computationally intensive when migrating the real dataset. On the other hand, the number of offsets is greater in synthetic dataset than in real one. As a consequence, the number of iterations of *Offset Loop* are greater in the synthetic dataset rather than real dataset. At last, remark that datasets migrated in production environment are more complex than these evaluated in this work. However, we believe that PKTM performance can be assessed with those datasets without loss of generality.

The PKTM performance was also evaluated with two different input parameters, since the variation of such parameters can significantly increase the PKTM execution time. The first parameter, FWIDTH, is used to define the high-end frequency increment for low-pass filter. Thus, the changing of FWIDTH reflects directly in the number of iterations performed by the *Filtering Loop*. The second one, NFC, defines the number of Fourier coefficients used for calculating low-pass filter values (line 4 in Algorithm 1). The execution time for calculating filter values increases as NFC grows. In other words, the higher the NFC and the lower the FWDITH, the greater the PKTM execution time is.

We evaluated the execution time of three PKTM versions: (i) the original Seismic Unix code; (ii) the same Seismic Unix code auto-vectorized by GCC; (iii) our PKTM implementation by using SSE. The auto-vectorized version was compiled with optimization flags *ffast-math* and *O3* which enables by default the auto-vectorization flags *ftree-slp-vectorize* and *ftree-vectorize*. Our implementation was also compiled with *-O3*, but the flags *fno-tree-vectorize* and *fno-tree-slp-vectorize* were used to disable auto-vectorization. Moreover, we do not support GCC to find out the proper loops to parallelize, i.e., the overall code analysis was lonely accomplished by GCC. All PKTM versions were compiled with GCC version 4.8.4 and the experiments were accomplished, without any external

[1] For confidentiality reasons, we can not inform the exact localization of this dataset.



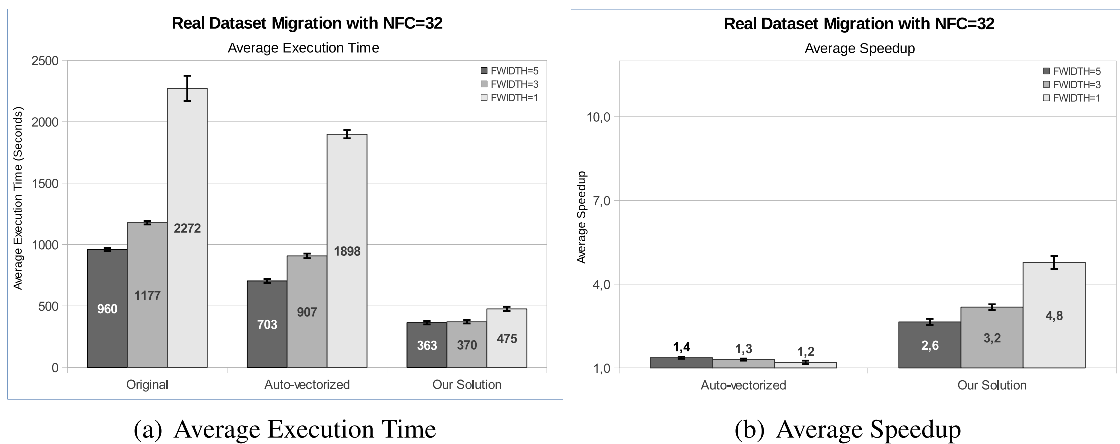
(a) Average Execution Time (b) Average Speedup

Figure 1. Synthetic Dataset Migration with NFC=32

interference (like processes and users), in a Intel i5-3337U 1.80GHz processor with 6GB of RAM memory running Ubuntu Linux (kernel 3.13.0). We executed the Seismic Unix version 43R8 for the following NFC and FWIDTH values: 32, 16, 8, and 5, 3, 1, respectively, testing all combinations of these values. The values chosen for NFC and FWIDTH obey an interval of feasible values that match to geophysical principles. Each experiment was repeated five times in order to calculate the average execution time achieved by each code. At last, we considered an error margin calculated from a 99% confidence interval by using the Student's T Distribution.

The average execution time achieved by the three PKTM codes as well as the speedups achieved by our solution and by the auto-vectorized code when migrating synthetic dataset with NFC equal to 32 are shown, respectively, in Figures 1a and 1b. As expected, the average execution time of all PKTM versions grows as FWIDTH decreases. For all FWIDTH values, the average execution time of our implementation was shorter than the other ones. When FWIDTH was equal to 1 the speedup achieved by our implementation was approximately eight times greater than the one achieved by the auto-vectorized version. This super-linear speedup comes from the adoption of SSE combined with another optimization process accomplished by GCC such as loop unrolling and function reordering to improve code locality, for example. Moreover, the speedup achieved by our solution increases as the PKTM execution time grows, indicating that the acceleration provided by our implementation follows the growth of PKTM complexity execution. However, unlike our implementation, the speedup of the auto-vectorized version is approximately constant, even when the PKTM complexity execution rises. Actually, note that the speedup slightly decreases as PKTM execution time increases.

The average execution time achieved by the three PKTM codes as well as the speedups achieved by our solution and by the auto-vectorized code when migrating real dataset with NFC equal to 32 are shown, respectively, in Figures 2a and 2b. When migrating the real dataset, both our implementation and the auto-vectorized one follow the same behavior when migrating the synthetic dataset, i.e., the speedup of our implementation increases as PKTM complexity grows, whereas the speedup of auto-vectorized version remains approximately constant. The main difference between migrating the synthetic and the real datasets is that, in the second one, the acceleration achieved by our imple-



(a) Average Execution Time (b) Average Speedup

Figure 2. Real Dataset Migration with NFC=32

mentation was smaller. However, even though the speedup achieved when migrating the real dataset was smaller than migrating the real one, the performance achieved by our solution is still approximately four times greater than the performance achieved by the auto-vectorized version. These results show that GCC compiler can not successfully vectorize PKTM. This happens because the auto-vectorization process performed by GCC are not designed to support some programming patterns such as loops with wrap around variables, besides that GCC auto-vectorization fail in the presence of data dependencies

5. Conclusion and Future Work

In this paper we proposed an accelerated version of PKTM by using SSE, a set of SIMD vector instructions available in off-the-shelf Intel processors. The experimental results considering both synthetic and real datasets showed that our solution is till nine times faster than the original one and up to eight times faster than the auto-vectorized version generated by GCC. These results show that GCC compiler can not successfully vectorize PKTM due to some programming patterns such as loops with wrap around variables and flow control which are presented in this algorithm.

In future work we expect to evaluate the performance achieved by a PKTM solution that combines GPU and CPU, where the CPU code, implemented by using SIMD vector instructions, is used to perform computation and not only control tasks. We also expect to implement PKTM by using a novel set of SIMD instructions such as AVX which allows to process 8 floating point numbers simultaneously. At last, although this is not the main goal of this paper, we intend to further evaluate commercial compilers such as ICC or frameworks like LLVM, as well.

References

Bryant, R. E., David Richard, O., and David Richard, O. (2003). *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River.

Cohen, J. and Stockwell, J. J. (2008). Cwp/su: Seismic unix release no. 41: An open source software package for seismic research and processing. *Center for Wave Phenomena, Colorado School of Mines*.

Deschizeaux, B. and Blanc, J.-Y. (2007). Imaging earth's subsurface using cuda. *GPU Gems*, 3:831–850.

Intel (2015). The intel intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. [Online; accessed in July-2015].

Levin, S. A. et al. (2004). Numerical precision in 3d prestack kirchhoff migration. In *SEG Annual Meeting*. Society of Exploration Geophysicists.

Maleki, S., Gao, Y., Garzaran, M. J., Wong, T., and Padua, D. A. (2011). An evaluation of vectorizing compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 372–382. IEEE.

Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., and Zhou, J. (2013). Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *27th International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW)*, pages 1107–1116. IEEE.

Panetta, J., Teixeira, T., de Souza Filho, P. R., da Cunha Filho, C. A., Sotelo, D., da Motta, F. M. R., Pinheiro, S. S., Rosa, A. L. R., Monnerat, L. R., Carneiro, L. T., et al. (2012). Accelerating time and depth seismic migration by cpu and gpu cooperation. *International Journal of Parallel Programming*, 40(3):290–312.

Pommier, J. (2007). Simple SSE and SSE2 (and now NEON) optimized sin, cos, log and exp. `http://gruntthepeon.free.fr/ssemath/`. [Online; accessed July-2015].

Rahmad, M. H., Meng, S. S., Karuppiah, E. K., and Ong, H. (2011). Comparison of cpu and gpu implementation of computing absolute difference. In *International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 132–137. IEEE.

Rizvandi, N. B., Boloori, A. J., Kamyabpour, N., and Zomaya, A. Y. (2011). Mapreduce implementation of prestack kirchhoff time migration (pktm) on seismic data. In *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 86–91. IEEE.

Shi, X., Li, C., Wang, S., and Wang, X. (2011). Computing prestack kirchhoff time migration on general purpose gpu. *Computers & Geosciences*, 37(10):1702–1710.

Sun, P. and Shi, X. (2012). An opencl approach of prestack kirchhoff time migration algorithm on general purpose gpu. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 179–183. IEEE.

Teixeira, D., Yeh, A., and Sampath Gajawada, T. (2013). Implementation of kirchhoff prestack depth migration on gpu. *SEG Technical Program Expanded Abstracts*, 3683:3686.

Xu, R., Hugues, M., Calandra, H., Chandrasekaran, S., and Chapman, B. (2014). Accelerating kirchhoff migration on gpu using directives. In *First Workshop on Accelerator Programming using Directives*, pages 37–46. IEEE.

Yilmaz, O. and Doherty, S. M. (1987). *Seismic Data Processing*, volume 2 of *Investigations in Geophysics*. Society of Exploration.