RC25068 (W1010-109) October 27, 2010
Computer Science

IBM Research Report

High Performance Computing of Line of Sight Viewshed

Ligang Lu, Brent Paulovicks, Vadim Sheinin, Michael Perrone
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

High Performance Computing of Line of Sight Viewshed

Ligang Lu, Brent Paulovicks, Vadim Sheinin, and Michael Perrone
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598 USA

Abstract

In this paper we present our recent research and development work for multicore computing of Line of Sight (LoS) on the Cell Broadband Engine (CBE) processors. LoS can be found in many applications where real-time high performance computing is required. We will describe an efficient LoS multi-core parallel computing algorithm, including the data partition and computation load allocation strategies to fully utilize the CBE's computational resources for efficient LoS viewshed parallel computing. In addition, we will also illustrate a successive fast transpose algorithm to prepare the input data for efficient Single-Instruction-Multiple-Data (SIMD) operations. Furthermore, we describe the data input and output (I/O) management scheme to reduce the (I/O) latency in Direct-Memory-Access (DMA) data fetching and storing operations. The performance evaluation of our LoS viewshed computing scheme over an area of interest (AOI) with more than 4.19 million points has shown that our parallel computing algorithm on CBE takes less than 2.3 ms, which is more than 15 times faster than the computation on an Intel x86 system.

1. Introduction

High performance real-time Line-of-Sight (LoS) viewshed computation has important applications in civil, defense, and military operations. In these applications, the visibility from an observation or target point to other points within an area of interest (AOI) needs to be determined. The map of all visible points in the AOI from that given point constitutes the LoS viewshed. In general, the problem of LoS viewshed computation is to determine the visibility of all points in an AOI from the given observation or target point. Some real-time applications often require the LoS viewshed computation to be performed faster than the real-time to save time for subsequent operations. The run-time performance of current available commercial software packages ESRI ArcGIS (3D Analysis/Viewshed) [3] and MicroDEM [4] are in order of magnitudes slower than the real-time. For examples, to compute viewshed over a square of 2047x2047 grid points, it takes ESRI ArcGIS 61,000 milliseconds and MicroDEM 17,000 milliseconds on a 2.4 GHz Core II Duo computer, respectively. Therefore new and efficient parallel computing algorithms for LoS viewshed are needed to enable and support faster than real-time application.

The calculation of the LoS viewshed involves computing the visibility of all the points in AOI. Determining the visibility of a point P_t takes two steps. The first step is to compute the coordinates of the points on the line between the observation point P_0 and P_t. In the discrete coordinate system, a line is represented by a zig-zag approximation as shown in Figure 1(a). We use IBM's well known Bresenham Line Drawing Algorithm to determine the coordinates of the points on the line. The second step is to calculate the slope in elevation between P_0 and P_t to determine the visibility of P_t, as shown in Figure 1(b). The slope is determined by the ratio of the difference in the elevation over the distance between P_t and P_0. If the slope is larger than the maximum slope among all previous points on the same line, then P_t is visible from P_0; otherwise P_0 is invisible. These two steps can be combined and simplified to optimize the viewshed computing. For significant size of AOI, real-time LoS viewshed calculation in general requires a high performance computer system. For convenience, we use the square shaped AOI. However the parallel computing algorithm we describe here can be extended to AOI with different shapes. The square AOI can be defined by a parameter R to represent a (2R+1) by (2R+1) square. The input elevation data is typically the Digital Elevation Model (DEM) images in GTIF file format.

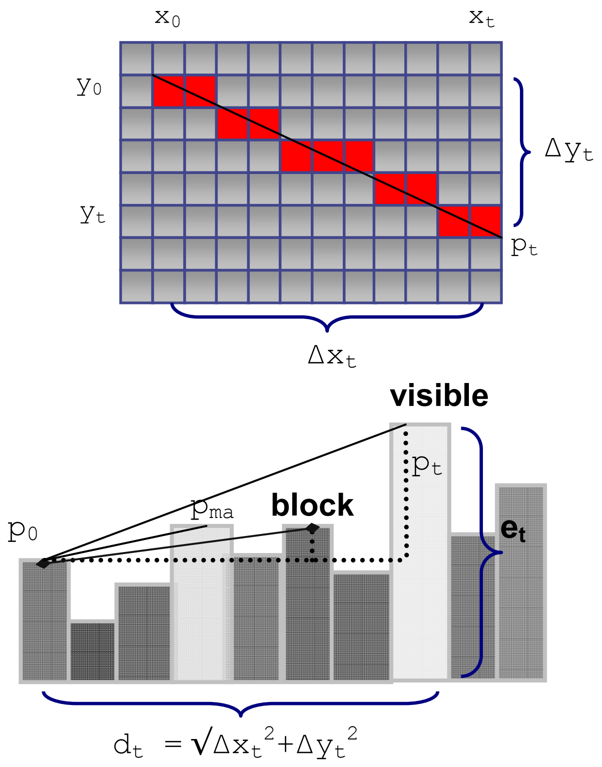


Figure 1. (a) Bresenham Algorithm to draw a line from point P_0 to P_t; (b) the Angle algorithm to calculate the elevation slope and compare it with the previous maximum slope to determine its visibility;

The Cell Broadband Engine (CBE) or Cell [2] in short was jointly developed by Sony, IBM, and Toshiba for Sony's PlayStation video game consoles. CBE is a heterogeneous multiprocessor chip consisting eight synergistic processing elements (SPEs) that are coordinated by a PowerPC Processing Element (PPE) (Figure 3). The SPEs execute 32-bit wide instructions in Single Instruction Multiple Data (SIMD) operations on four parallel 32-bit words and 128-bit registers. The PPE also has 128-bit vector processing capability. All the elements are connected to an internal Element Interconnect Bus (EIB). An on-chip Memory Interface Controller (MIC) is also connected to the EIB. SPEs and

PPE access the system main storage by asynchronous Direct Memory Access (DMA) through MIC and the Rambus XDR IOs. Such asynchronous DMA design provides high speed data transfer with a total bandwidth of 25.6 GB/s and concurrent computation operations. The Cell Broadband Engine Interface (BEI) provides communications with the rest of the system through the high speed Flex IO interface. In short, CBE is designed for running computation intensive SIMD applications with large data volume; it provides a very impressive computing power with an aggregate performance of 204.8 Giga FLOPS at a 3.2 GHz operating frequency.

In this paper, we will present our new work on developing better than real-time parallel computing of LoS viewshed on Cell. Specifically, we will describe an efficient parallel computing algorithm, including data partition scheme, successive byte shuffle algorithm for data transpose for SIMD opeartion, and data input and output managing strategies to leverage the computing power and throughput of the Cell computer to efficiently calculate LoS viewshed. We will also show an exemplary client-server system for the application. Our test results have shown that our Cell based LoS computing algorithm can perform more than 15 times faster than that on Intel's x86 system.

The rest of the paper is organized as the follows. In Section 2, we will describe the LoS viewshed parallel computing algorithm on Cell, including data partition, computation load distribution, and SIMD parallel computing in SPEs; then in Section 3, we will illustrate a successive byte shuffle algorithm to transpose data for SIMD; in Section 4, we will present our data input and output strategies to minimize the overall latency; and In Section 5 we will show the test performance of our algorithm.

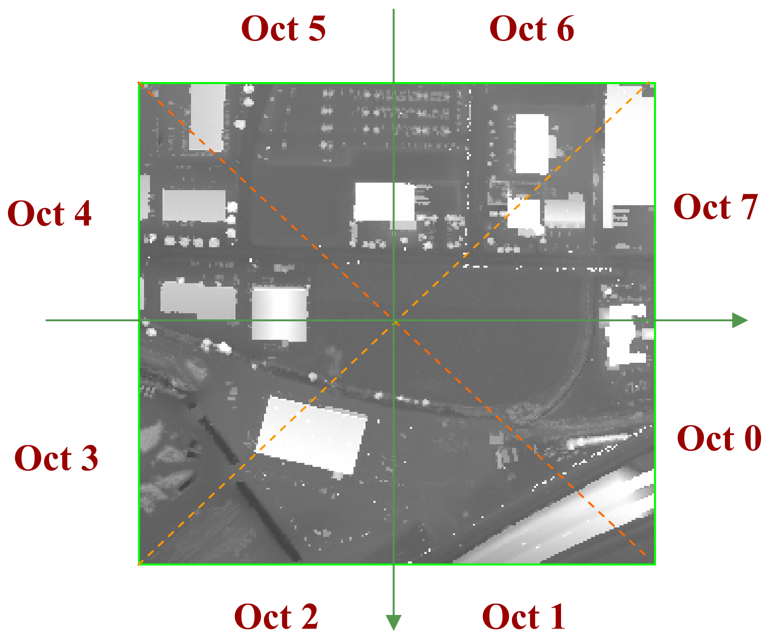


Figure 2. An example of the AOI data partition scheme

2. LoS Viewshed Parallel Computing on CBE

One important issue in parallel computing algorithm design for LoS viewshed application is how to partition the data and distribute the computing load of the LoS problem to efficiently utilize the resource and HW/SW capabilities of CBE. The data partition and computation load allocation will directly impact the LoS viewshed computing performance. In this section we will describe our strategies and schemes on data partition and work load distribution.

2.1 Data Partition Strategy for Parallel Computing

From the LoS algorithm description in the previous section, we can see that the visibility of a point on a line depends on the maximum elevation slope of all previous points. To maximize the LoS parallel computing efficiency, we need to partition the data into sub-areas such that each sub-area can be processed independent from the other sub-areas. Furthermore, to fully utilize the parallel computing resources and capability of a CBE with 8 SPEs and 1 PPE, we also need to allocate the computation load wisely among the processors. Since PPE is designed for launching SPEs and data IO operations between the CBE and the external storage, it is more suitable for the control logic, i.e., playing the managing and bookkeeping roles.

For the given AOI, we partition it into sub-areas in such a way that a ray coming out from the observation point will intersect with only one sub-area as the ray grows. Figure 2 shows an example of our data partition method. We partition the AOI of a (2R+1) by (2R+1) square into 8 octants; the lines of sight grow out of the center observation point will not intersect with more than one octant. Such partition can enable the independent computing of each and every octant by a processor while the redundancy in data fetching and store is minimized.

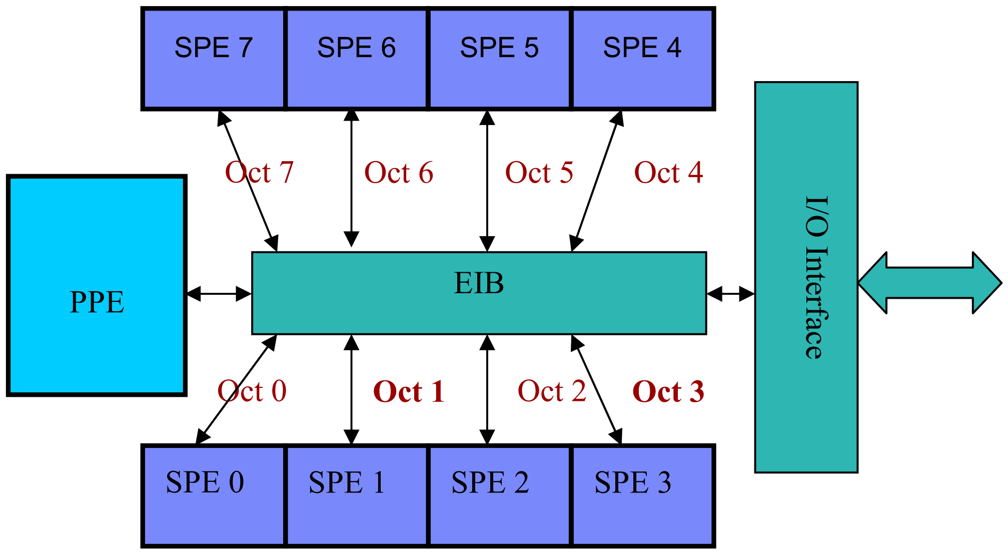


Figure 3. LoS viewshed computation load distribution on CBE

2.2 Computation Load Distribution

We use 8 SPEs and 1 PPE of the CBE to do the LoS viewshed parallel computation work. To make efficient use of the characteristics of SPE and PPE, we assign each SPE to compute the LoS viewshed for one octant of the AOI while use PPE for the administrative work, such as preparing the parameters and passing them to SPEs, initiating the tasks on SPEs, and assembling the results from SPEs. Figure 3 illustrates the computation load partition in our LoS viewshed parallel computing scheme.

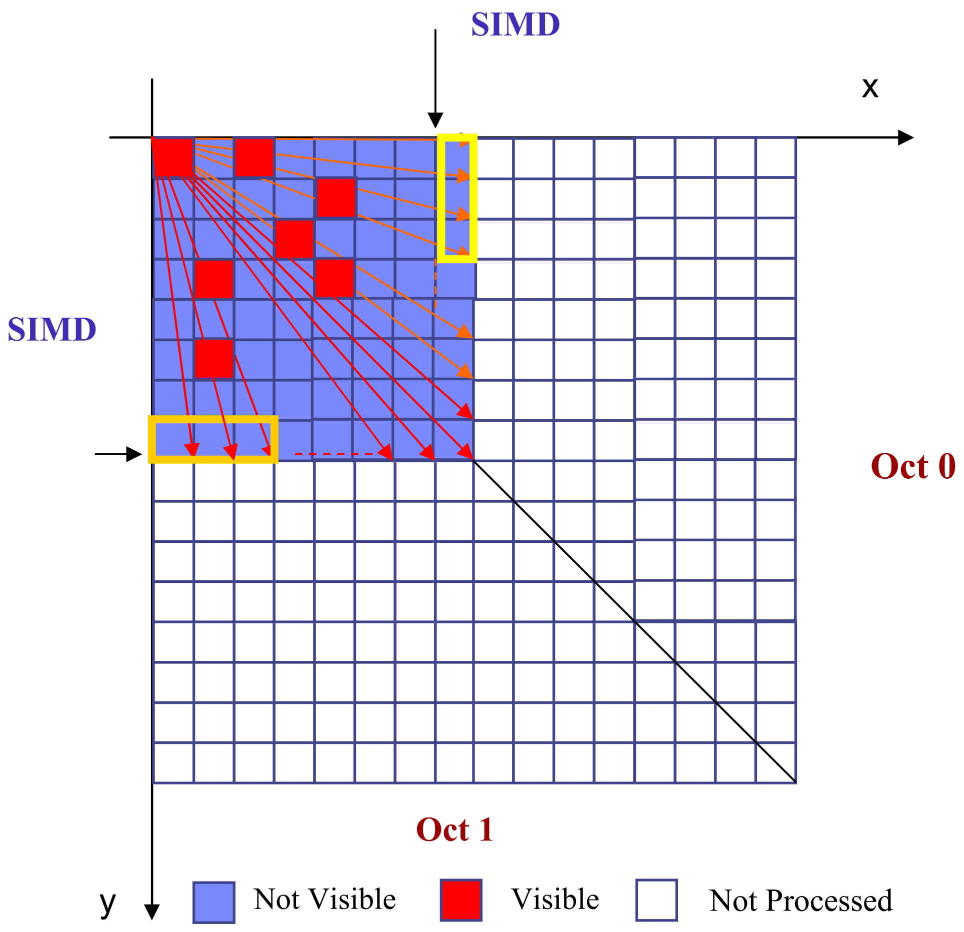


Figure 4. Parallel computing using SIMD operations in SPE

3 Parallel Computing in SPEs

One very important aspect of our LoS parallel computing scheme is about how to efficiently utilize the single-instruction-multiple-data (SIMD) capability, also known as vector processing. Because the visibility of a point on a ray has dependency on the maximum slope of all previous points on that ray, parallel computation using SIMD operations can not be done on multiple points along the same ray. However, we can apply SIMD operations on multiple points across the rays. For example as shown in

Figure 4, in Octant 0, we need to grow R+1 rays from the center point where x=0 to the edge of the AOI where x = R. For each step in x, we use SIMD operations to compute the visibilities of the points of 4 rays at the same x (as the yellow box shown in Figure 4) one step at a time, until the visibilities of all rays at that x (only y varies) are computed. In Octant 1, the same parallel computing scheme can be applied with x and y coordinates exchanged.

From Figure 4, it can be seen that in order to apply SIMD in Octant 0, the DEM data of a 4-point column needs to be loaded into the instruction register. Since the physical memory is designed for data to be read from and write to in rows not in columns, it would require 4 memory-reads to fetch the data followed by several other instructions to put the needed bytes into the register. Such process would be very inefficient. We devised and implemented a novel scheme by fetching in the data in multiple rows and transposing the data using an efficient successive byte shuffle algorithm.

	p0	**p1**	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**
v0	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v1	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v2	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v3	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v4	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v5	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v6	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*
v7	*D0*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*	*D7*

(a)

	p0	**p1**	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**
v0	*D0*	*D1*	*D2*	*D3*	*D0*	*D1*	*D2*	*D3*
v1	*D0*	*D1*	*D2*	*D3*	*D0*	*D1*	*D2*	*D3*
v2	*D0*	*D1*	*D2*	*D3*	*D0*	*D1*	*D2*	*D3*
v3	*D0*	*D1*	*D2*	*D3*	*D0*	*D1*	*D2*	*D3*
v4	*D4*	*D5*	*D6*	*D7*	*D4*	*D5*	*D6*	*D7*
v5	*D4*	*D5*	*D6*	*D7*	*D4*	*D5*	*D6*	*D7*
v6	*D4*	*D5*	*D6*	*D7*	*D4*	*D5*	*D6*	*D7*
v7	*D4*	*D5*	*D6*	*D7*	*D4*	*D5*	*D6*	*D7*

(b)

	p0	**p1**	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**
v0	*D0*	*D1*	*D0*	*D1*	*D0*	*D1*	*D0*	*D1*
v1	*D0*	*D1*	*D0*	*D1*	*D0*	*D1*	*D0*	*D1*
v2	*D2*	*D3*	*D2*	*D3*	*D2*	*D3*	*D2*	*D3*
v3	*D2*	*D3*	*D2*	*D3*	*D2*	*D3*	*D2*	*D3*
v4	*D4*	*D5*	*D4*	*D5*	*D4*	*D5*	*D4*	*D5*
v5	*D4*	*D5*	*D4*	*D5*	*D4*	*D5*	*D4*	*D5*
v6	*D6*	*D7*	*D6*	*D7*	*D6*	*D7*	*D6*	*D7*
v7	*D6*	*D7*	*D6*	*D7*	*D6*	*D7*	*D6*	*D7*

(c)

	p0	**p1**	**p2**	**p3**	**p4**	**p5**	**p6**	**p7**
v0	*D0*	*D0*	*D0*	*D0*	*D0*	*D0*	*D0*	*D0*
v1	*D1*	*D1*	*D1*	*D1*	*D1*	*D1*	*D1*	*D1*
v2	*D2*	*D2*	*D2*	*D2*	*D2*	*D2*	*D2*	*D2*
v3	*D3*	*D3*	*D3*	*D3*	*D3*	*D3*	*D3*	*D3*
v4	*D4*	*D4*	*D4*	*D4*	*D4*	*D4*	*D4*	*D4*
v5	*D5*	*D5*	*D5*	*D5*	*D5*	*D5*	*D5*	*D5*
v6	*D6*	*D6*	*D6*	*D6*	*D6*	*D6*	*D6*	*D6*
v7	*D7*	*D7*	*D7*	*D7*	*D7*	*D7*	*D7*	*D7*

(d)

Fig. 5. Successive Byte Shuffling Algorithm for Data Transpose (a) Original 8x8 Data: 8 vectors each with 8 data elements; (b) After byte shuffles between v0 and v2, v1 and v3, v4 and v6, v5 and v7, respectively; (c) After byte shuffles between v0 and v4, v1 and v5, v2 and v6, v3 and v7, respectively; (d) After byte shuffles between v0 and v1, v2 and v3, v4 and v5, v6 and v7, respectively.

3.1 Data Transpose Using Successive Byte Shuffling Algorithm

We have devised an algorithm to transpose data using efficient SIMD byte shuffle operations successively. Fig. 5 illustrates how the algorithm works by an example of transposing an 8x8 block of data. The elements of the 8x8 data block are loaded into 8 vector variables or registers, v0, v1, …, v7, and each of the vectors holds 8 data elements in position p0, p1, …, p7, respectively, as shown in Fig. 5 (a). In this case, the algorithm completes the transpose in three successive steps with SIMD byte shuffle operations.

Step 1: Perform a 4x4 data block swap in the reverse diagonal direction, i.e., the data elements in position p4~p7 in v0~v3 are swapped with the elements in p0~p3 in v4~v7. Fig. 5 (b) shows the results after the SIMD byte shuffle operations of this step.

Step 2: Perform 2x2 data block swaps, i.e., data elements in p2~p3 and p6~p7 in v0~v1 and v4~v5 are swapped with the data elements in p0~p1 and p4~p5 in v2~v3 and v6~v7, respectively. Fig. 5(c) shows the results after the SIMD byte shuffle operations of this step.

Step 3: Perform 1x1 data block swaps, i.e., data elements in p1, p3, p5, and p7 in v0, v2, v4, and v6 are swapped with the data elements in p0, p2, p4, and p6 in v1, v3, v5, and v7, respectively. Fig. 5(d) shows the results after the SIMD byte shuffle operations of this step.

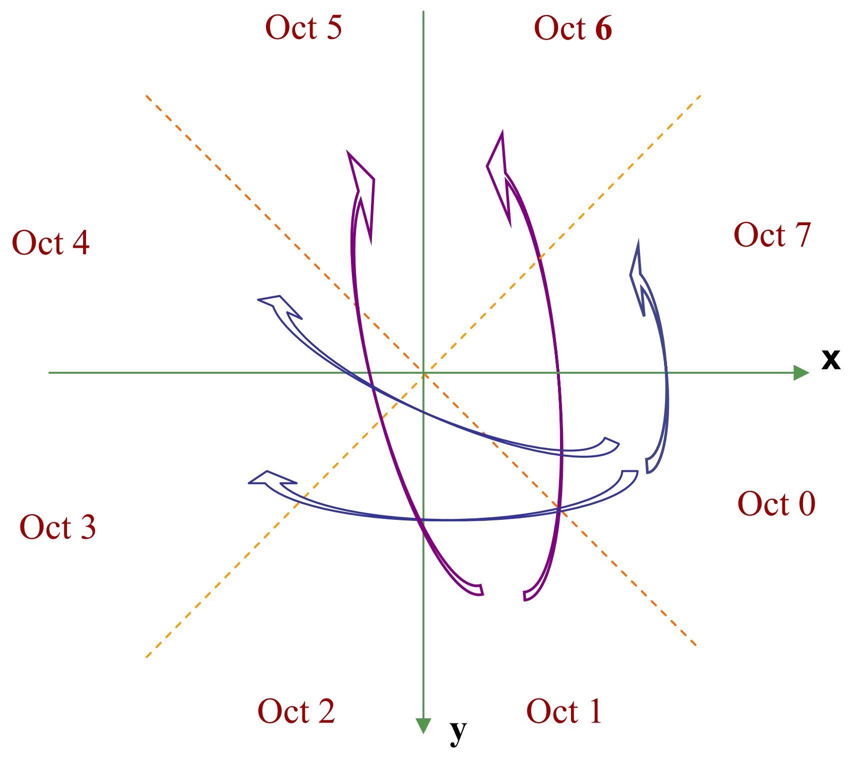


Figure 6. Reflections of other octants with respect to Octant 0 or Octant 1

Once we developed efficient schemes to compute Octant 0 and Octant 1 using SIMD operations, the other octants can be treated as various reflections of Octant 0 or Octant 1 and processed similarly as shown in Figure 6. Specifically, Octants 3, 4, and 7 can be computed using the same method for Octant 0 with data transpose and special attention to negative axis increment. Octants 2, 5, and 6 can be viewed as one of the flips of Octant 1 and processed accordingly.

4. DMA and Time-line Management

Another important aspect in efficient parallel computing is the memory access and task time-line management. The SPEs access the system main memory through the Direct Memory Access (DMA) operations. Because memory access requires certain time cycles to fetch or store data, if data is not available when a processor needs the data to process, it has to waits for the DMA operation to complete and becomes idle before it can proceed. This will result in the loss of computing cycles. The objective of DMA operation time-line management is to wisely parallelize the memory access operations with the computing processing to hide the DMA wait time and minimize or even eliminate the processor's idle time so that the parallel computation performance can be maximized. In this section, we will present our DMA double buffering scheme and their time-line arrangement aligned with the computation process in order to achieve the best attainable performance.

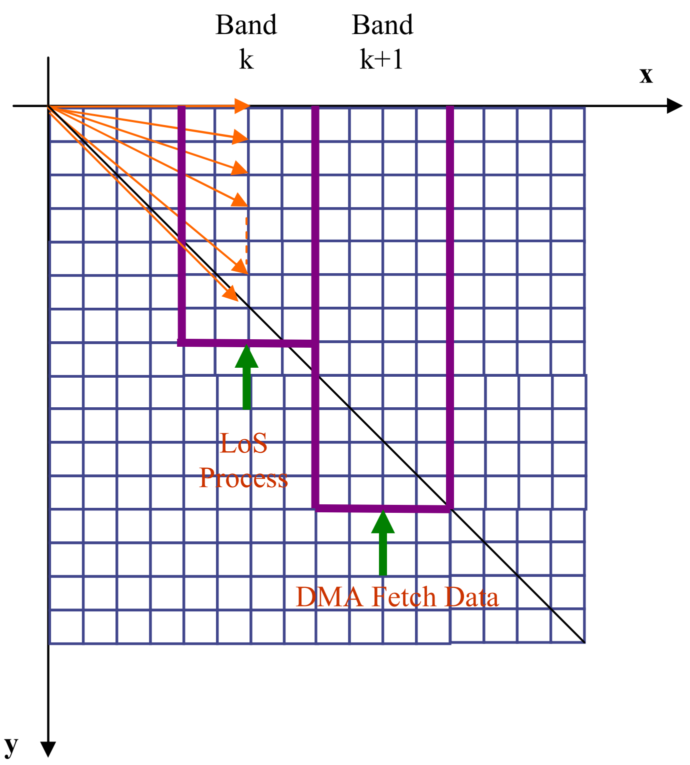


Figure 7. DMA double buffer scheme to minimize wait time

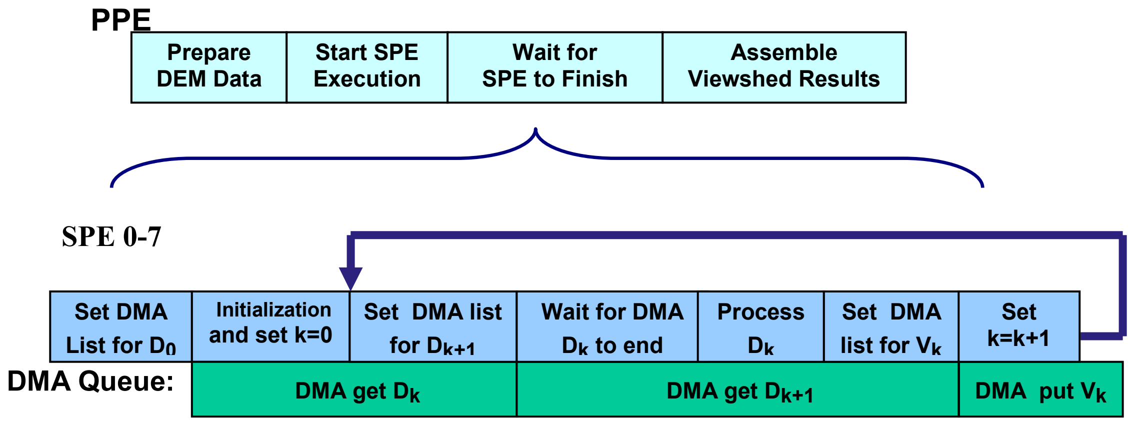
4.1 DMA Double Buffering Scheme

To hide the DMA operation time as much as possible, we devised a double buffering scheme to parallelize the operations of DMA and LoS viewshed computing. Figure 7 illustrates the double buffer scheme that parallelizes the operations of LoS computing and DMA data fetching of a SPE operating on an octant. The octant is partitioned into bands of columns with certain width. The entire band of data is fetched from the system memory into the SPE local store efficiently using the DMA list operation. While the SPE processor is computing the LoS visibilities on the data in Band k that has been already read in and transposed in one data buffer, it is also fetching the DEM data of Band k+1 into another data buffer into the SPE's local store. In the double buffering scheme, the data computing process and the data fetch or store processes are organized into parallel processes. We used DMA double buffering schemes for the DEM data fetches as well as the viewshed result output stores and almost all DMA operation time have been hidden. The total SPE processors' idle time has been reduced to negligible cycle numbers.

For the octants that do not need data transposed, we also employed the DMA double buffering schemes for fetching DEM input data and storing viewshed output to effectively minimize the processor wait time.

4.2 Data Processing and DMA Operation Time-line Management

To minimize the processor's idle time, the order and the alignment of the data computation process and DMA process are also extremely important. The order of these processes need to be carefully planed and managed because the arrangement of the execution orders of these processes in the time-line will directly impact the over performance. Figure 8 depicts the time-line orders for our LoS parallel computing on CBE. As it can be seen, the DMA operations are mostly in parallel with the data computing process.



- D_k : DEM Data Band k;
- V_k: Viewshed Result for Band k

Figure 8. LoS data processing and DMA operation alignment in time-line

5. Performance

In order to facilitate the research and development for the LoS viewshed application, we developed a Graphic User Interface (GUI) tool for displaying the DEM image and viewshed results. Figure 9 shows the GUI is displaying a DEM image; the AOI is a square in the green frame with a size of (2R+1)x(2R+1) and R is specified by the user. The calculated viewshed of this AOI is also displayed on the GUI with the visible points marked in red. The system we developed for LoS viewshed application is a server-client system as shown in Figure 10. The server is running on a Cell Blade QS21 with 8 Synergistic Processing Elements (SPEs) and one PowerPC Processing Element (PPE). It also has main memory and file system installed. On the client side is an Intel x86 computer (a desk-top or laptop PC) with the GUI installed for DEM data viewing and the computed viewshed displaying. The server and the client are connected through a high data transport link.

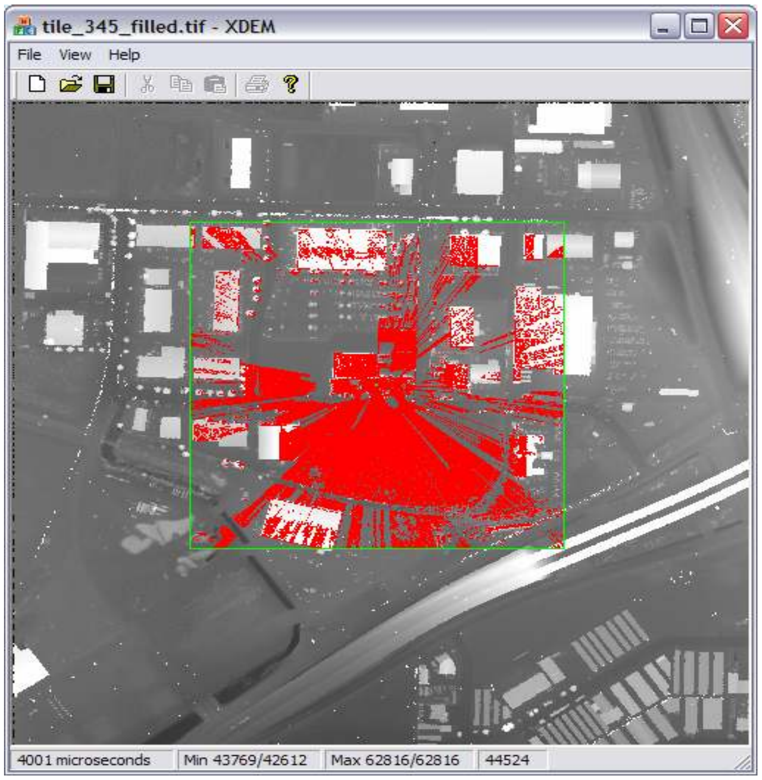


Figure 10. Line of Sight AOI and computed viewshed displayed on GUI

In the process, the user at the client side selects an AOI in the DEM image displaying on the GUI window and sends a request for LoS viewshed calculation to the server. The DEM data in the specified AOI on the GUI window will be sent to the server. The server then computes the LoS viewshed and sends the viewshed back to the client for overlay displaying in the GUI window.

Table 1 presents the LoS time performance on CBE QS21 we have achieved. For comparison, we also implement the LoS viewshed calculation on an Intel 2 Duo CPU@2.2 GHz. For (2R+1)x(2R+1) =2027x2027, the LoS performance on Cell is ~15x faster than that on the x86.

Table 1. LoS Performance on CBE (Time in Milliseconds)

R	**SPE 0**	**SPE 1**	**SPE 2**	**SPE 3**	**SPE 4**	**SPE 5**	**SPE 6**	**SPE 7**	**PPE Proc**	**Cell Total**	**x86**
255	**.300**	**.248**	**.286**	**.300**	**.300**	**.281**	**.250**	**.300**	**1.495**	**2.037**	**21.499**
511	**1.216**	**0.903**	**1.019**	**1.249**	**1.223**	**1.002**	**0.907**	**1.272**	**5.729**	**7.276**	**95.380**
1023	**5.266**	**3.213**	**3.692**	**5.324**	**5.354**	**3.678**	**3.212**	**5.330**	**19.809**	**25.421**	**378.81**

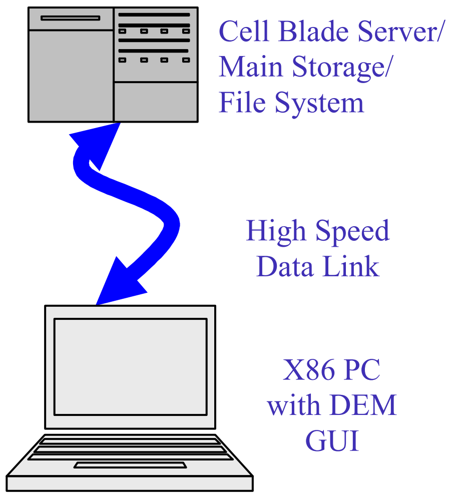


Figure 10. Server and client configuration for LoS viewshed system

6. Conclusions

In this paper we presented our work on high performance computing of LoS viewshed based on CBE. We described our efficient parallel computing algorithm by utilizing the HW resources and computation power of the CBE. We presented our strategies in data partition and work load distribution, and illustrated our data transpose algorithm using successive byte shuffling for efficient SIMD operations in SPEs. In addition, we showed how to timely align the LoS computing process and DMA IO operations to reduce the latency. Our results have shown that our LoS viewshed computation on CBE achieved faster than run-time performance.

Reference

[1]. J.E. Bresenham, “Algorithm for computer control of a digital plotter,” IBM Systems Journal, vol. 4, no.1, January 1965, pp. 25–30.

[2] “Cell Broadband Engine,” Book IV for DD1.0, Version 1.0, SCIE/Toshiba/IBM, May, 2004.
[3] http://www.esri.com/software/arcgis/arcgisengine/extension/3danalyst/index.html
[4] http://www.usna.edu/Users/oceano/pguth/website/microdem/microdem.htm