ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

Implementation of Digital Signal Processing Algorithm in General Purpose Graphics Processing Unit (GPGPU)

Jagdamb Behari Srivastava[1], R.K. pandey[2], Jitendra Jain[3]

Programme Asst. (Computer), Jawaharlal Nehru Krishi Vishwavidyalaya , Jabalpur, India[1]

Dr. , Prof. & Head, UICSA, Rani Durgavati Vishwavidyalaya , Jabalpur, India[2]

Assistant Professor, Dept. of ECE, JUET College of Engineering, Guna, India[3]

ABSTRACT: In this paper, we have proposed sequential and parallel matrix and matrix-vector multiplication in compute unified device architecture (CUDA) libraries. We show the process of a class of algorithms parallelization which are used in digital signal processing. We present this approach on the instance of the Linear Convolution, Circular Convolution, and Least Mean Square (LMS) algorithm. We propose an approach which uses a general purpose graphics processor unit (GPGPU) technology. The accelerated version on GPU computed faster because it took less time compare to the MATLAB and sequential implementation.

Keywords: Linear Convolution, Circular Convolution, Least Mean Square (LMS), Computed unified device architecture (CUDA).

I. INTRODUCTION

A graphics processing unit (GPU) provides a parallel architecture which combines raw computation power with programmability. The architecture of GPU offers a large degree of parallelism at a relatively low cost through the well know vector processing model know as Single Instruction, Multiple Data (SIMD) [1], [2]. A computer which exploit multiple data stream against a single instruction stream to perform operations which may be naturally parallelized. For example, array processor and GPU [3]. There are two types of processor used in GPU i.e. Vertex and Pixel (or fragment) processors. The programmable Vertex and Pixel processors execute a user defined assembly level program having 4-way SIMD instructions. The Vertex processor performs mathematical operations that transform a vertex into a screen position. This result is then pipelined to the Pixel or fragment processor, which performs the texturing operations [4]. GPU can only process independent Vertices and Pixels processor, but can process many of them in parallel.

In this sense, GPU are stream processors. Each element in the stream are applied function using kernels. There are many of algorithms uses in digital signal processing which need a strong computational power to work in real time. In many situations the most complex (from the computational point of view) part of these algorithms is problem of large matrix multiplication and matrix-vector multiplication. In this paper, primary focus has been given to the matrix multiplication and matrix-vector multiplication.

MATLAB is a high-level language and interactive computation environment that enables to user to perform computationally intensive tasks [5]. This paper presents the speed-up achieved and benefits of the CUDA accelerated version of Linear Convolution, Circular Convolution and Least Mean Square (LMS) algorithm.

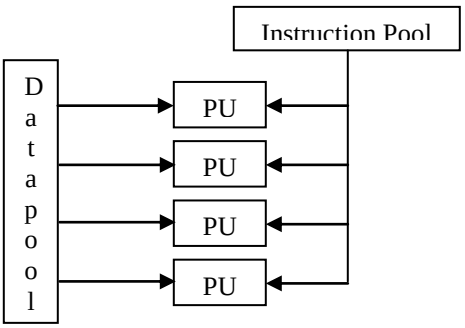


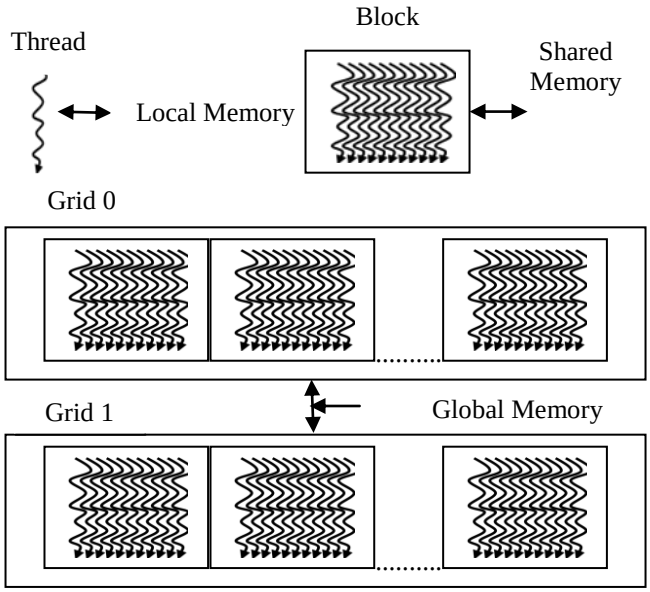
Figure 1: Single Instruction, Multiple Data (SIMD).
Physical Unit (PU)

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

II. MEMORY MANEGMENT

Global memory resides on the device, but off-chip from the multiprocessors, so that access times to global memory can be 100 times greater than to shared memory. All threads in the kernel have access to all data in global memory. Shared memory is comparable to L1 cache memory on a regular CPU.



It resides close to the multiprocessor, and has very short access times. Shared memory is shared among all the threads of a given block. Thread-specific memory stored where global memory is stored. Variables are stored in a thread's local memory if the compiler decides that there are not enough registers to hold the thread's data. This memory is slow, even though it's called "local" [6].
The main problem in parallelization of existing algorithms on CUDA devices is appropriate use of memory. On NVIDIA Tesla architecture, a thread block has 16 kB of shared memory visible to all threads of the block. All threads have access to the same global memory. Global memory can be 100 times greater than to shared memory. When there is no bank conflicts accessing the shared memory are fast as accessing a register. For comparison access to the global memory takes 400 -600 cycles [7].

III. PARALLEL PROCESSING BASED ON CUDA

CUDA was introduced by NIVIDIA Company in November 2006. It is a general purpose parallel computing architecture with a new parallel programming model, and an instruction set architecture that leverages the parallel compute engine in NVIDIA GPU to solve many complex scientific computational problems in a more efficient way than on CPU [6].
The CUDA program model includes a host and one or more devices. The host's execution unit is CPU which executes the serial program, because CPU has a strong power to do logic operation. And the device's execution unit is GPU which executes the parallel program because GPU has many arithmetic logic units (ALU) processing at the same time. The program which executed in the device is called kernel which is executed N times in parallel by the different N CUDA threads. In CUDA program model, the thread is the least computed unit. The kernel is executed in each thread. A certain number of kernels can consist a block. The maximum number of kernel in a block is 512. Every thread in a block can share the shared memory in block.

IV. DIGITAL SIGNAL PROCESSING ALGORITHM

- Linear Convolution vs. Circular Convolution

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

Consider a finite duration sequence $x(n)$ of length N_1 which is given a input to an finite impulse response (FIR) system with impulse response $h(n)$ of length N_2, then the output is given by

$$y(n) = x(n) * h(n) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

$$= \sum_{k=0}^{N_1-1} x(k)h(n-k), n = 0,1,\ldots\ldots N_1 + N_2 - 1 \ldots\ldots\ldots(2)$$

$$= \sum_{k=0}^{N_1-1} h(k)x(n-k), n = 0,1,\ldots\ldots N_1 + N_2 - 1 \ldots\ldots\ldots(3)$$

Where,

$x(n)$ = 0, n < 0 and $n \geq N_1$

$h(n)$ = 0, n < 0 and $n \geq N_2$

The N-point circular convolution of $x(n)$ with $h(n)$ must be equivalent to the linear convolution of $x(n)$ with $h(n)$. In general, the linear convolution of two sequences $x(n)$ and $h(n)$ with lengths N_1 and N_2 respectively will give a sequence with a length of $N \geq N_1 + N_2 - 1$. When $x(n)$ and $h(n)$ have a duration less than N, for implementing linear convolution using circle convolution, N_2 - 1 and N_1 - 1 zeros are padded (added) at the end of $x(n)$ an $h(n)$ respectively to increase their length to N.

- Least Mean Square (LMS) Algorithm

We know that the most complex part of the LMS algorithm is matrix multiplication part [8]. The main advantage of LMS algorithm is that it requires simple computation. In practical applications of adaptive filtering, a fixed step size algorithm. In this filter we assume that a modification of $\Delta h(n)$ vector of the $h(n)$ filter parameter should be proportional in each time moment n to the cost function, gradient vector $j(n)$, which can be written an equation

$$h_M(n+1) = h_M(n) + \Delta h_M(n) \ldots\ldots\ldots \ldots\ldots\ldots\ldots ..(4)$$

$$h_M(n+1) = h_M(n) - \frac{1}{2} u(n) \frac{\partial j(h(n))}{\partial h(n)} \ldots\ldots\ldots(5)$$

Where $u(n)$ is a scale variable which influences onto the speed of the filter modification. To speed up of the adaptation process, an additionally weight matrix is introduced. Such modified equation (5) takes the form of:

$$h_M(n+1) = h_M(n) - \frac{1}{2} u(n) w(n) \frac{\partial j(h(n))}{\partial h(n)} \ldots\ldots\ldots(6)$$

In the case of LMS a temporal error value is minimized. Therefore the error criterion takes the form of :

$$j = e^2(n) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(7)$$

Finally, the equation (4) takes the form of :

$$h_M(n+1) = h_M(n) + u(n)w(n)e(n)x(n) \ldots\ldots\ldots \ldots\ldots\ldots(8)$$

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

This can be formulated in the matrix form as:

$$\begin{bmatrix} h_0(n+1) \\ \vdots \\ h_{M(n+1)} \end{bmatrix} = \begin{bmatrix} h_0(n) \\ \vdots \\ h_{M(n)} \end{bmatrix} + u(n) \begin{bmatrix} W_{1,1}(n) & \cdots & W_{1,1}(n) \\ \vdots & \ddots & \vdots \\ W_{1,M+1(n)} & \cdots & W_{M+1,M+1(n)} \end{bmatrix} \quad ..e(n) \begin{bmatrix} x(n) \\ \vdots \\ x(n-M) \end{bmatrix} \ldots\ldots\ldots (9)$$

V. GPU PROGRAMMING

The code was implemented in Microsoft Visual Studio 2010 (Express Edition) C with CUDA 5.0.

- Kernels

A kernel is the unit of work that the main program running on the host computer offloads to the GPU for computation on that device. In CUDA, launching a kernel requires specifying three things:

- The dimensions of the grid
- The dimensions of the blocks
- The kernel functions to run on the device.

The main matrix (square) code is presented below.

```
__global__ void gpu_matrixmult(int *gpu_a, int *gpu_b, int *gpu_c, int N) {
  int k, sum = 0;
  int col = threadIdx.x + blockDim.x * blockIdx.x;
  int row = threadIdx.y + blockDim.y * blockIdx.y;

  if(col < N && row < N) {
  for(k = 0; k < N; k++)
      sum += gpu_a[row * N + k] * gpu_b[k * N + col];
              gpu_c[row * N + col] = sum;
          }
}
// ALLOCATE MEMORY AND LOAD

size = N * N * sizeof (int);
          a = (int*) malloc(size);
          b = (int*) malloc(size);
          c = (int*) malloc(size);
// ALLOCATE MEMORY ON DEVICE

          cudaMalloc((void**)&dev_a,size); cudaMalloc((void**)&dev_b, size);
          cudaMalloc((void**)&dev_c, size);
// LOAD ARRAYS WITH NUMBERS

          loadArrays(a,b,c,N);                                    printf("Array A\n");
printArray(a, N);
          printf("Array B\n"); printArray(b, N);
//COMPUTATION DONE ON GPU TIME

cudaMemcpy(dev_a, a , size ,cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b , size ,cudaMemcpyHostToDevice);
          cudaEventCreate(&start);
          cudaEventCreate(&stop);
          cudaEventRecord(start, 0);
// KERNELS CALL
```

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

```
            dim3 Grid(Grid_Dim_x, Grid_Dim_y);
            dim3 Block(Block_Dim_x,Block_Dim_y);
gpu_matrixmult<<<Grid,Block>>>(dev_a,dev_b,dev_c,N);
            cudaEventRecord(stop, 0);
            cudaEventSynchronize(stop);
            cudaEventElapsedTime(&elapsed_time_ms1, start, stop );
printf("Time to calculate results on GPU: %f ms.\n", elapsed_time_ms1);
  cudaMemcpy(c,dev_c, size ,cudaMemcpyDeviceToHost);
//CLEAN UP

            free(a);
            free(b);
            free(c);
            cudaFree(dev_a);
            cudaFree(dev_b);
            cudaFree(dev_c);
            cudaEventDestroy(start);
            cudaEventDestroy(stop);
                  getch();
            return(0);
}
}
```

The main matrix vector code is presented below.

```
__global__ void gpu_matrixmult( int *a, int *b, int *c,int N)
{
            int sum=0,k;
  int col =  blockIdx.x * blockDim.x + threadIdx.x;
  int row =0;
      if(col < N && row < N) {
                        for(k = 0; k < N; k++)
                        sum += a[col * N + k] *b[k];
                  c[row * N + col] = sum;
              }
}
```

VI. SIMULATION RESULTS

There were two versions of algorithm sequential and parallel. Both of them was coded in C with using CUDA libraries and run on NVIDIA GeForce GT 630M with 2 GB dedicated VRAM (Video Random Access Memory) graphics card installed on Acer V3-571G, Intel Core i5-3210M 2.5GHz with Turbo Boost up to 3.1 GHz.

We have implemented of sequential matrix multiplication, sequential matrix-vector multiplication, parallel matrix multiplication and parallel matrix-vector multiplication for different matrix dimensions, shows the result in Table I and Table II respectively.

The computation of MATLAB and CUDA together with GeForce GT 630M graphics card solved the day to day increasing demand for the massive parallel general purpose computing. The accelerated version on GPU was astonishingly fast, because it took less time compare to the MATLAB and sequential implementation. However, this value is strictly limited to the execution of computation kernel itself and doesn't include any overhead cost. Comparison between proposed implementation and Ing.et.al. [9]for an dimensional matrix multiplication, show the result in Table III.

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

Table I: Comparison between sequential and parallel matrix multiplication for different matrix dimensions.

Matrix Size (Square)	GeForce GT 630M	CPU
3×3	0.009152	0.432
4×4	0.009664	6.782
8×8	0.011360	15.012
16×16	0.016256	47.00
32×32	0.039296	110.00
64×64	0.132032	266.00
128×128	1.045248	795.00
256×256	9.401632	3135.00

Table II: Comparison between sequential and parallel matrix-vector multiplication for different matrix dimensions.

Matrix	Vector	GeForce GT 630M	CPU
3×3	3×1	0.013152	0.432
4×4	4×1	0.019664	1.782
8×8	8×1	0.023360	3.012
16×16	16×1	0.018256	5.013
32×32	32×1	0.049344	8.124
64×64	64×1	0.091392	12.158
128×128	128×1	0.189888	15.498
256×256	256×1	0.448288	22.510

Table III: Comparison between proposed implementation and Ing. Vaclav et. al.,[9] for an 512×512 dimensional.

	Runtime(sec)
MATLAB implementation	0.949422
Sequential implementation	0.786238
Ing. Vaclav et. al.,[9]	0.548716
Proposed implementation	0.076442

VII. CONCLUSIONS

In this paper, we implemented and studied the performance of computing the absolute difference using matrix multiplication and matrix-vector multiplication. It has been shown that proposed GPU implementation better than other GPU, MATLAB, sequential implementation. We have shown that GPU is an excellent candidate for performing data intensive digital signal algorithms. A direct application of this work is to perform Linear Convolution, Circular Convolution and Least Mean Square (LMS) algorithm on GPU in real time digital signal processing.

REFERENCES

[1] K. Fatahalian and M. Houston, "GPUs: A Closer Look", ACM Queue,Vol. 6, No. 2, March/April 2008, pp. 18–28

[2] C.J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis", in Proc. 35th International Sympo. Microarchitecture, pp. 306-317, 2002.

[3] I. Bucks, "Data Parallel Computing on Graphics Hardware", Graphics Hardware 2003 Panel: GPUs as Stream Processors, July 27, 2003.

[4] J. D. Owens, et. al, "A Survey of General-Purpose Computation on Graphics Hardware", *Eurographics 2005*, August 2005, pp. 21-51.

[5] S. P. Mohanty, N. Pati, and E. Kougianos, "A Watermarking Co-Processor for New Generation Graphics Processing Units", in *Proc. 25th IEEE International Conference on Consumer Electronics*, pp. 303-304, 2007.

[6] NVIDIA CUDA Compute Unied Device Architecture, 2007. Programming Guide.

ISSN(Print):2320–9798
ISSN (Online): 2320 – 9801

International Journal of Innovative Research in Computer and Communication Engineering
Vol. 1, Issue 4, June 2013

[7] S. Che, J. Meng, J. W. Sheaer, and K. Skadron. A Performance study of general purpose applications ongraphics processors. In First Workshop on General Purpose Processing on Graphics Processing Units, page 10.
[8] W. Bozejko, M. Walczynski, M. Wodecki, Application beam- search algorithm based on fast Fourier transform for signal analysis (in polish), Automatyka, Zeszyty Naukowe Politechniki Sl„!skiej, Gliwice 2008, z.150, pp.31-38.
[9] Ing. Vaclav Simek and Ram Rakesh ASN, "GPU Acceleration of 2D-DWT Image Compression in MATLAB with CUDA",in IEEE Second UKSIM European Symposium on Computer Modeling and Simulation, pp. 978-0-7695-3325-4/08, 2008.