Applications of Berkeley's Dwarfs on Nvidia GPUs

HPSC-14 **Yang Zhang and Haiqing Wang**

1 Introduction

In recent years, general purpose computing on GPUs has become more and more a trend in scientific computations since APIs like OpenCL or Nvidia's CUDA make the development process easier. GPUs provide very high parallelism and for the achieved performance they are relatively cost-efficient. Thus, in this report we keep the main messages from several articles which focus on applications implemented by using CUDA on Nvidia GPUs. Furthermore, the applications fit in the different dwarfs devised by the University of California, Berkeley [7].
The report is structured as followed: In Section 2, an overview of the hardware architecture and the programming model CUDA is presented. Starting with the dynamic programming article [1], which will be explored more in detail, section 3 introduces five of the thirteen dwarfs and their related articles. It explains which specific problem each article tackles and shows the main idea aswell as the results of them. Lastly, section 4 gives an overall statement and finishes up with a conclusion.

2 GPU Architecture and Programming Model

Fermi Architecture

We have chosen to introduce the Fermi architecture (Figure 1) since most of our papers were using GPUs with that architecture. The first Fermi GPUs were released in 2010 and featured over 3 billion transistors and up to 512 cores divided by 16 streaming multiprocessors (SM). Fermi has a so-called GigaThread global scheduler which distributes thread blocks to streaming multiprocessor's thread schedulers. One SM (Figure 2) consists of 32 cores, over 32K 32-bit registers, 16 load/store units which allow the computing of source and destination addresses for 16 threads in one clock, four special function units (SFU) for instructions such as trigonometric functions, 64 KB on-chip memory configurable between shared memory and L1 cache, two warp scheduler aswell as two instruction dispatch units. The host interface connects the GPU to the CPU through PCI-Express bus. Fermi supports up to 6 GB of GDDR5 DRAM memory because of the 64-bit adressing capability and has a common L2 cache of 768 KB [9].
We want to indicate that Fermi is only the third-latest of Nvidia's architectures with Kepler being the successor to it and Maxwell being the latest one. However, Nvidia already planned the next architecture called Pascal due in 2016.

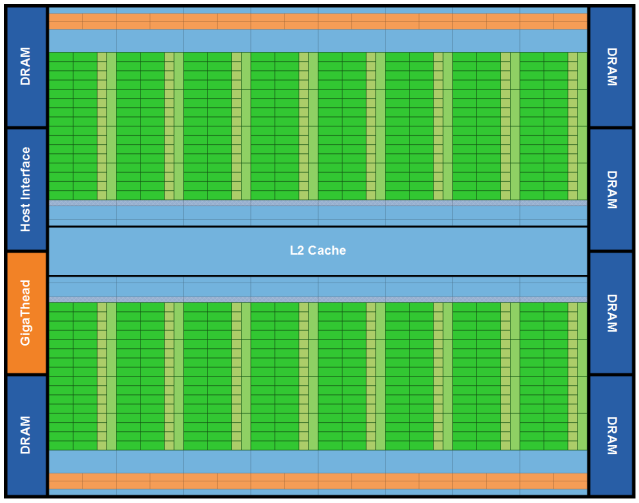


Figure 1: Overview of the Fermi architecture [9]

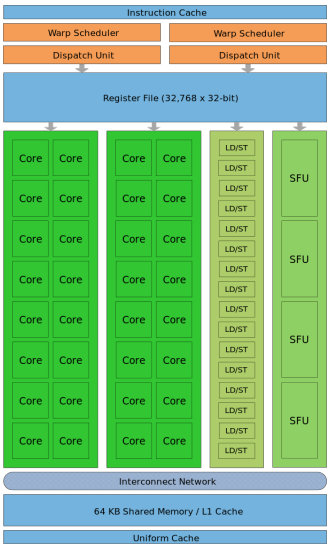


Figure 2: A Fermi Streaming Multiprocessor [9]

CUDA

CUDA, first released in 2007 by Nvidia, is a parallel computing platform and a programming model for general purpose computing on GPUs (GPGPU). It supports various programming languages including C/C++ and Fortran. Furthermore, a lot of libraries are available e.g. cuSPARSE for sparse matrix operations, cuBLAS for linear algebra operations or NPP for sound and image processing and many others. This section presents a brief overview of the processing flow and the execution model of CUDA.

Figure 3 shows an illustration of the processing flow on CUDA. First the processing data is copied from the main memory to the GPU memory, then the CPU instructs the GPU to execute the function (kernel) on the GPU. The GPU starts executing the kernel in each core concurrently. Finally, the GPU stores the results in its memory which is then copied back to the main memory.

In CUDA each thread has an ID assigned. A group of threads build a block and a group of blocks build a grid. A thread is executed by a core and a block is executed by a SM. Kernels are launched as grids and are executed one at the time. When a block is assigned to a SM, it is split further into so-called warps consisting of 32 threads each. Moreover, blocks are independent of each other and can be executed in parallel. As for the memory allocation, each thread has its own private local memory allocated while each block has a shared memory. Threads belonging to the same block are able to communicate with each other due to the shared memory. Besides the access to global memory, each thread also has access to a constant memory i.e. a read-only memory.

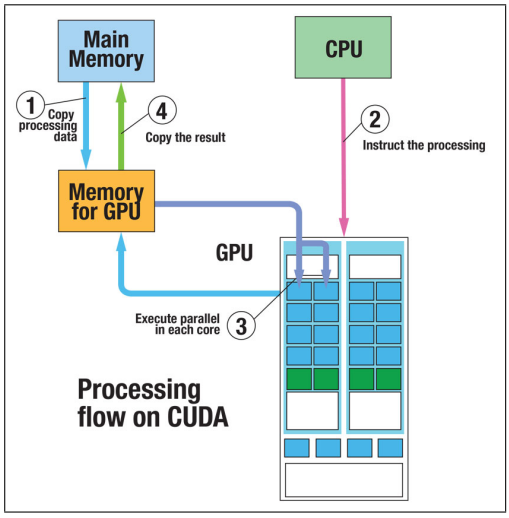


Figure 3: CUDA Processing Flow

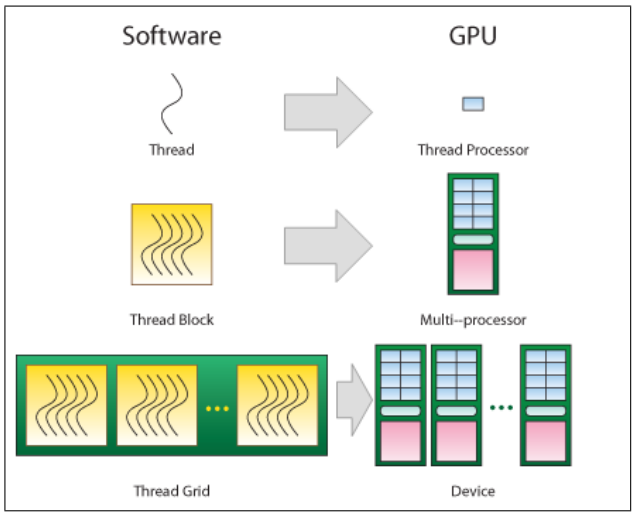


Figure 4: CUDA Execution Model

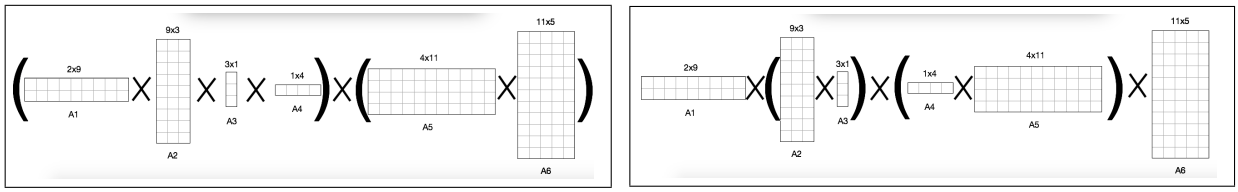
3 The Dwarfs

3.1 Dynamic Programming

The Dynamic Programming Dwarf is about an algorithmic technique that compute solutions by solving simpler overlapping subproblems.
The paper I found talks about the Matrix Chain Product problem which can be solved by dynamic programming and how it uses CUDA to implement parallelism.

Matrix Chain Product

First of all, the product of an $l \times m$ matrix and an $m \times n$ matrix needs $l \cdot m \cdot n$ multiplications and the size of the resulting matrix is $l \times n$. Matrix Chain Product Problem is an optimization problem for finding the best parentheses of the matrix chain that minimizes the total number of multiplications [1]. For example, given 6 matrices A_1, A_2, A_3, A_4, A_5 and A_6 with size 2×9, 9×3, 3×1, 1×4, 4×11 and 11×5. Figure 5 shows an example of two different parentheses with the results 328 and 221 respectively.



(a) The result is 328 (b) The result is 221

Figure 5: An example of different parentheses

Dynamic Programming Algorithm

Suppose the matrix chain is $\langle A_1, A_2, ..., A_n \rangle$. Let $P = \langle p_0, p_1, ..., p_n \rangle$ be a sequence of dimensions of matrix A_i for i = 1,2,...,n, such that the size of A_i is $p_{i-1} \times p_i$. Let $m_{i,j} (1 \leq i \leq j \leq n)$ denote the minimum number of multiplications to compute the product of $\langle A_i, A_{i+1}, ..., A_j \rangle$. The goal is to compute the $m_{1,n}$ and to find relative parentheses.
Firstly, it should be clear that $m_{i,i} = 0$ because one matrix dose not need any multiplication.
And then it should also be clear that $m_{i,i+1} = p_{i-1} \cdot p_i \cdot p_{i+1}$ because $m_{i,i+1}$ means the direct product of A_i and A_{i+1}.
Generally, to compute the $m_{i,j}$, we should know the solutions of its subproblems, that is to compute $m_{i,k}$ and $m_{k+1,j}$ $(i \leq k < j)$ in advance, and select the best one. So

$$m_{i,j} = \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j)$$

$p_{i-1} \cdot p_k \cdot p_j$ means the number of multiplications when producting the result matrix of $\langle A_i, ..., A_k \rangle$ and the result matrix of $\langle A_{k+1}, ..., A_j \rangle$. Also,

$$s_{i,j} = arg \min_{i \leq k < j} (m_{i,k} + m_{k+1,j} + p_{i-1} \cdot p_k \cdot p_j)$$

it is used to record the best k. The whole algorithm is showed as Figure 6.
Figure 7 illustrates an example of talbes m and s for the same matrix chain in Figure 5, A_1, A_2, A_3, A_4, A_5 and A_6 with size 2×9, 9×3, 3×1, 1×4, 4×11 and 11×5.

Matrix Chain Product Algorithm
1. $m_{i,i} \leftarrow 0$ for every i $(1 \leq i \leq n)$
2. $m_{i,j} \leftarrow \infty$ for every i and j $(1 \leq i < j \leq n)$
3. for $l \leftarrow 2$ to n do
4. for $i \leftarrow 1$ to $n - l + 1$ do
5. $j \leftarrow i + l - 1$
6. for $k \leftarrow i$ to $j - 1$ do
7. $q \leftarrow m_{i,k} + m_{k+1,j} + p_{i-1} p_k p_j$
8. if $q < m_{i,j}$ then $m_{i,j} \leftarrow q$, $s_{i,j} \leftarrow k$

Figure 6: Matrix Chain Product Algorithm [1]

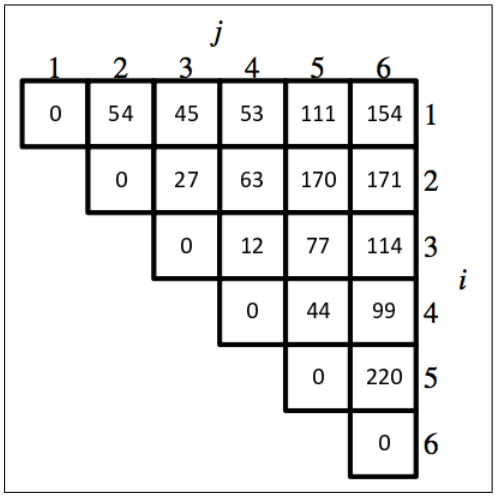


Figure 7: An examples of table m (n = 6) [1]

GPU Implementation

Dynamic Programming algorithm for Matrix Chain Product Problem can be parallelized in two ways. One way is the loop in lines 4-8 of Figure 6, which computes the costs in the diagonal entries of tables m and s in Figure 7, e.g. when l=4 it computes $m_{1,4}, s_{1,4}, m_{2,5}, s_{2,5}, m_{3,6}$ and $s_{3,6}$. These calculations do not use current diagonal entries, instead they only use values from smaller l. So the computing is independent that is it can be done in parallel. Another way is the loop in lines 6-8 of Figure 6, which computes one entry of tables m and s, e.g. to get $m_{1,4}$, firstly compute $(m_{1,3} + m_{4,4} + p_0 \cdot p_3 \cdot p_4)$,$(m_{1,2} + m_{3,4} + p_0 \cdot p_2 \cdot p_4)$ and $(m_{1,1}, m_{2,4} + p_0 \cdot p_1 \cdot p_4)$ and then select the minimum one. Obviously, each $(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)(1 \leq k < 4)$ can be computed

independently. So it can be done in parallel.
Figure 8 shows the amount of the computation for each l. In CUDA, it is not easy to obtain good parallelization performance since the performance depends on various factors, for example, Occupancy, a number of Streaming Multiprocessors, amount of computation, and so on [1]. So we use three different types of parallelism: OneThreadPerOneEntry, OneBlockPerOneEntry, and BlocksPerOneEntry. The combination of them can make performance better.
OneThreadPerOneEntry, this kernel allocates one Thread to compute one entry in the tables m and s. The execution in the lines 6-8 in Figure 6 is done by one Thread [1]. For example, $\langle m_{1,4}, m_{2,5}, m_{3,6} \rangle$, each one is computed concurrently by one Thread.
OneBlockPerOneEntry, this kernel allocates one Block to compute one entry in the tables m and s. The execution in lines 6-8 in Figure 6 is done by several Threads in one Block [1]. For example, $m_{1,4} = \min_{1 \leq k < 4}(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$, the whole thing is computed by one Block. So each $(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$ is computed by one Thread, and one more Thread is to select the minimum value.
BlocksPerOneEntry, this Kernel allocates multiple Blocks to compute for one entry. This Kernel consists of two Kernel calls to synchronize between Blocks. One is to compute the costs by Blocks in parallel. The other is to obtain the minimum one [1]. For example, $m_{1,4} = \min_{1 \leq k < 4}(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$, the whole thing is computed by two or more Blocks. So each $(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$ is computed by one Thread but maybe from different Blocks, and one more Thread from some Block is to select the minimum value.

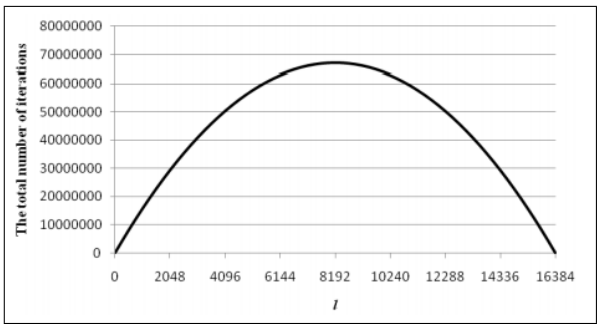


Figure 8: The computation amount for each l [1]

(a) Execution time of OneThreadPerOneEntry [ms]

Threads per Block	32	64	128	256
	729,428	977,575	1,017,610	1,112,098

(b) Execution time of OneBlockPerOneEntry [ms]

Threads per Block	32	64	128	256
	912,642	1,111,794	1,137,166	1,056,523

(c) Execution time of BlocksPerOneEntry [ms]

	Blocks per Entry			
Threads per Block	2	4	8	16
32	1,571,230	1,561,411	1,493,823	1,338,796
64	1,251,733	1,221,975	1,115,941	1,001,370
128	1,215,875	1,059,025	970,862	1,038,715
256	1,128,318	989,715	1,095,406	1,447,205

Figure 9: Total time of each kernel [1]

Evaluation

From Figure 9, we can know different number of threads per block and different number of blocks per entry have different performance when n = 16384. For first two kernels, 32 threads per block has the best performance. For the third kernel, 128 threads per block and 8 blocks per entry has the best performance. Figure 10 illustrates the fastest kernel in different range of l ,from which we can combine these three kernels to get better performance. Figure 11 shows the total running time of GPU (combination of three kernels) and CPU. Here, GPU is Nvidia GeForce GTX 480 with 480 processing cores (15 Streaming Multiprocessors which has 32 processing cores) 1.4GHz, 3GB memory. And CPU is Intel Core i7 870, 2.93GHz, 8GB memory. But, the implementation from CPU is sequential program in C language. So the comparison is not really fair although the speed-up factor is up to about 41.

Range for l	Fastest Kernel
$2 \leq l \leq 3668$	OneBlockPerOneEntry
$3669 \leq l \leq 15406$	OneThreadPerOneEntry
$15407 \leq l \leq 15662$	OneBlockPerOneEntry
$15663 \leq l \leq 16384$	BlocksPerOneEntry

Figure 10: Fastest Kernel for different l [1]

GPU[s]	CPU[s]	Speed-up
701.6	29,282	41.7

Figure 11: Running time of GPU and CPU [1]

3.2 Sparse Linear Algebra

The Sparse Linear Algebra Dwarf mainly describes operations with matrices where the entries consist of a large amount of zeros. The sparse matrices are usually stored in specific formats to reduce the storage (e.g. DIA, CRS, ELL, COO, etc...) [6]. The most commonly studied operation is the sparse matrix-vector product (SpMV), however in this section we focus on accelerating the sparse matrix-matrix product (SpMM) on GPU [2].

F. Vazquez et al. [2] proposes a new approach called FastSpMM based on the ELLR-T algorithm, which is a SpMV algorithm introduced in [8]. FastSpMM uses the ELLPACK-R storage format which is a variant of the ELLPACK storage format to fit the GPU environment. ELLPACK-R basically consists of three arrays called $A[]$, $J[]$ and $rl[]$. $A[]$ stores the entries, $J[]$ their column index and $rl[]$ the length of each row without the zeros. With FastSpMM every reading of an element in $A[]$ is related to the computation of several output elements [2].

Evaluation

For the evaluation of FastSpMM, two other versions of SpMM, namely ELLR-T and the level 3 function of the cuSPARSE library have been compared to it on Nvidia's GTX480 and Tesla C2050 GPUs. The ELLR-T version computes SpMM as a set of SpMV operations. The cuSPARSE routine uses the CRS storage format. The results for the GTX480 can be seen in Figure 12, the results for the Tesla C2050 (Figure 20) are very similar except that it reaches lower GFLOPs and can be seen in the appendices. Also, the characteristics of the various test matrices (Figure 21) and of both GPUs (Figure 22) are in the appendices aswell. As we can see FastSpMM outperforms the other two routines by a large factor on most test matrices.
Next, we compare the runtimes of FastSpMM on the GPU to the runtimes of a CPU implementation, which is from the MKL library, on an Intel Xeon E5640 with four cores. The results can be observed in Figure 13. We can conclude that for the GTX480 the speedup factor ranges from 2.8× to 6.2× and for the Tesla C2050 from 1.7× to 3.8×.

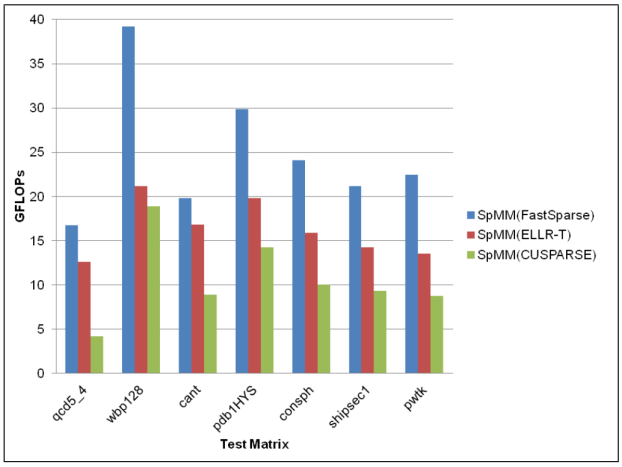


Figure 12: Performances of SpMM on GTX480 [2]

Matrix	N	GTX480	Tesla	MKL(4 cores)
qcd5_4	49152	15.12	24.95	41.6
wbp128	16384	6.09	10.00	37.6
cant	62451	30.04	39.38	139.5
pdb1HYS	36417	16.01	27.38	84.8
consph	83334	49.81	91.25	273.8
shipsec1	140874	157.38	212.97	567.3
pwtk	217918	396.45	389.90	1367.6

Figure 13: Runtimes (in seconds) of the two GPUs and the CPU [2]

3.3 Unstructured Grids

The Unstructured Grids Dwarf is about some problems which can be described in the form of updates on an irregular mesh or grid, with each grid element being updated from neighboring grid elements.
The paper I found talks about Compressible flows simulation on 3-D unstructured grids. Compressible flows is fluid mechanics that deals with flows having significant changes in fluid density. Figure 14 is an example that subsonic flow past a sphere with different number of elements. The number of elements means the number of 3-d unstructured grids. And this problem can be solved by Discontinuous Galerkin (DG) method. DG method in mathematics forms a class of numerical methods for solving differential equations. Also, DG method can use OpenACC-based program to be implemented in parallel. OpenACC (for Open Accelerators) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

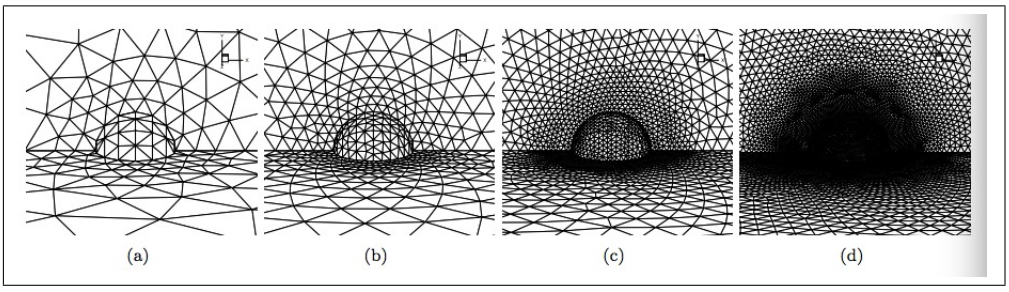


Figure 14: An example : Subsonic Flow past a Sphere [3]

Evaluation

GPU is NVIDIA Tesla K20c GPU containing 2496 multiprocessors which is implemented by OpenACC-based program ,and CPU is AMD Opteron 6128 CPU containing 16 cores which is

implemented by MPI-based parallel program.
Figure 15 illustrates the unit running time of GPU and CPU with different number of elements when implementing the simulations of the example showed in Figure 14. Nelem means the number of elements. CPU-1 means using only one core and CPU-16 means using all 16 cores. Obviously, performance of CPU-16 and GPU is much better than CPU-1. Generally, GPU has a better performance than CPU-16 except first situation. The reason of first situation is that there are some other stuffs like initialization, data transformation and file dumping which occupies much of running time and CPU is better at dealing with these stuffs [3].

	T_{unit} (microsecond)			Speedup	
Nelem	GPU	CPU-1	CPU-16	vs. CPU-1	vs. CPU-16
2,426	20.2	176.8	14.8	8.8	0.73
16,467	10.7	182.8	12.6	17.0	1.18
124,706	9.3	182.8	13.0	19.6	1.40
966,497	8.8	198.9	13.1	22.6	1.49

Figure 15: Timing measurements for subsonic flow past a sphere [3]

3.4 Combinational Logic

The Combinational Logic Dwarf revolves around simple operations, performed on a massive scale, which typically exploit bit-level parallelism [7]. Fitting examples would be the encryption/decryption of data, Cyclic Redundancy Codes or checksum computations. This section is dedicated the design of a parallel AES, a symmetric cryptographic algorithm, on GPU [4].

With the AES block cipher, the plaintext is split in 128-bit blocks and transformed in 10 to 14 rounds depending on the key size. Each round, with the exception of the first and the last round, consists of four steps which operate on an array of four 32-bit words, also called the state of the cipher [4]. The four steps are AddRoundKey (XOR addition), SubBytes(byte substitution), ShiftRows (cyclic shifting of bytes) and MixColumns (linear transformation). For the parallel AES there are two design choices, either fine-grained or coarse-grained. The fine-grained approach focuses on thread-level parellelism i.e. there is a lot of communication between the threads and also synchronization since each output of a round is the input of the next one. The coarse-grained approach focuses on higher-level parallelism and encrypts/decrypts whole blocks in parallel.

Evaluation

For the evaluation, the throughput of the fine-grained approach is compared to the coarse-grained one and a comparison between the GPU implementation and the CPU one is also considered. In Figure 16 we see that on a Nvidia 8800GT (112 cores) the fine-grained approach performs better on smaller plaintext sizes. The break-even point is at around 2 MB, at that point coarse-grained approach is the better choice. Similar results are seen with the Nvidia 8400GS (16 cores) except that the break-even point is at 512 KB. The main reasons for those results are that when using the coarse-grained approach on small plaintext sizes, there is a lot of load imbalance and when using the fine-grained approach on larger plaintext sizes, there is a lot of communication and synchronization

overhead. The fine-grained approach is better suited for practical applications since in practice, the plaintext size usually ranges from 35 KB to 150 KB [4].
Two Intel CPUs, the Core 2 Quad Q6600 and the Xeon Clovertown E5335 (both 4 cores) using the OpenSSL toolkit are compared to the aforementioned Nvidia GPUs as shown in Figure 17. The GPU implementation outperforms the CPU one on each plaintext size except for the 8400 GS where it performs slightly worse than the Core 2 Quad in the two smaller plaintext sizes. This is mainly due to the higher clock rate of the CPU. Also, when comparing the cost, we notice that the GPUs are by far cheaper than the CPUs.

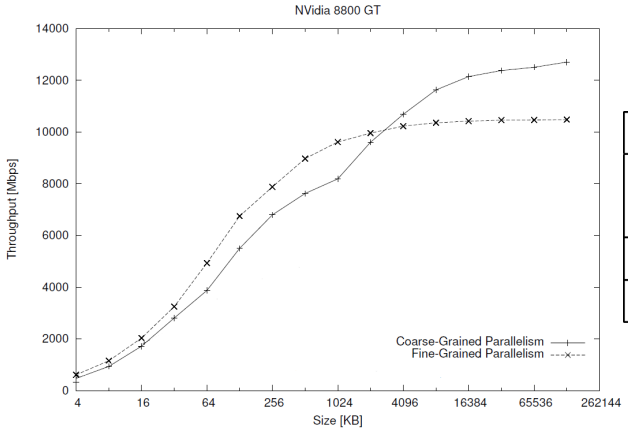


Figure 16: Throughput on Nvidia 8800 GT [4]

Plaintext Size	NVIDIA 8800 GT	NVIDIA 8400 GS	Intel Core 2 Quad	Intel Xeon E5335
32 KB	2917	771	862	721
128 KB	6591	855	868	724
32 MB	12075	908	857	709
128 MB	12412	909	856	708
Processor Specifications	112 SP	16 SP	64-bit mode	64-bit mode
Clock Freq. [MHz]	1500	900	2400	2000
Cost [$]	∼170	∼30	∼180	∼900

Figure 17: GPU vs. CPU Comparisons [4]

3.5 Graphical Model

The Graphical Model Dwarf is about a graph in which nodes represent variables, and edges represent conditional probabilities.
The paper I found talks about Speech Recognition System which uses two different graphic models, one is ANN (Artificial Neural Network) model and another is HMM (Hidden Markov Model). ANN model is for recognizing an acoustic fragment to a word. HMM is for combining recognized words. And training process of ANN model can be accelerated by GPU.
Figure 18 shows an ANN model, the input is a vector representing an acoustic fragment, and output is another vector representing the relative word. For example, giving an acoustic fragment with the meaning "dog" and the output word is "cat", we can use difference between "dog" and "cat" to train this model until the output word is "dog". In graph, each node in one layer is a value in relative vector and edges are unknown weights (parameters) to be trained. To make things simple, weights in two layers make up a weight vector so that inner product of a layer vector and a weight vector is the next layer vector. So, input vector inner products first weight vector equals hidden vector, and hidden vector inner products second weight vector equals output vector. Actually, transforming weight vector to its relative special matrix so that the effect of inner product of weight vector is same as multiplying its relative matrix. A lot of input vectors make up a big matrix. And a lot of output vectors make up another matrix. The input matrix multiply first weight matrix (transformed from first weight vector) equals hidden matrix, and hidden matrix multiply second weight matrix (transformed from second weight vector) equals output matrix. Input matrix and output matrix is giving, the goal is to optimize the two weight matrices, which can be solve by linear algebra. Especially, CuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines) library is

good at solving linear algebra problem.

Evaluation

The environment of implementation is 1600 MHz FSB, 8 GB RAM, NVIDIA GTX280 GPU with 240 cores which uses CuBLAS library and a quad-core 3.0 GHz CPU which uses Intel MKL library. Intel Math Kernel Library (Intel MKL) is a library of optimized math routines for science, engineering, and financial applications. Figure 19 shows the training time and relative speed-up for the WSJ0 corpus. The first implementation is standard sequential C program which is obvious the slowest one. And multi-thread MKL is a little better than single thread MKL which use single thread per core. CUDA is much better than others. Padding is about memory mapping which can make IO process faster. From the comparison of CUDA with padding and single thread MKL, we can get a speed-up factor of about 5.

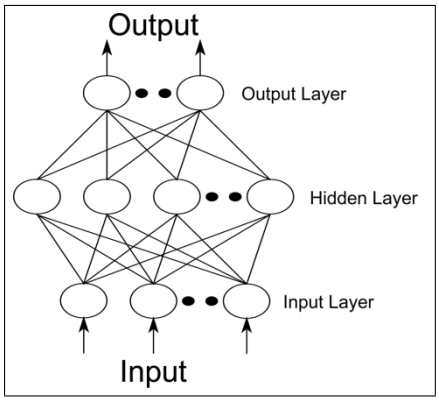


Figure 18: ANN Model

Training Implementation	Time hh:mm	Speed-up vs Standard C	Speed-up vs MKL
Standard C	42:35	-	-
Single thread MKL	11:36	3.7 x	-
Multi-Thread MKL	9:41	4.4 x	1.2 x
CUDA no-padding	2:29	17.1x	4.7x
CUDA with padding	2:14	19.1 x	5.2 x

Figure 19: Training time, and relative speed-up for the WSJ0 corpus [5]

4 Conclusion

In this report we have seen that Nvidia GPUs together with its CUDA programming model provide extremely high parallelism. Therefore, they are able to accelerate scientific computations by a considerable factor. Furthermore, GPGPU achieves high performance for low cost as seen in Figure 17 where the cost of the GPUs are much lower than the CPUs. GPGPU also reduces the CPU workload so that it enables the CPU to perform additional tasks. When we consider the learning curve, we would say that it looks rather smooth since standard languages like C/C++ are supported. However, the programmer must have precise knowledge about the underlying hardware architecture to bring out the full potential of the GPU. Lastly, we want to think about how simple it would be to implement an operation such as $x := \alpha x + y$ where α is a scalar and x and y two vectors. We presume that this is a fairly simple task, since it basically is the C implementation with a few added keywords and some CPU/GPU memory management.

Credits

In this report the workload was divided just as same as in the presentation.
Yang Zhang was responsible for the following sections: Introduction, GPU Architecture and Programming Model, Sparse Linear Algebra, Combinational Logic, Conclusion.
Haiqing Wang was responsible for the following sections: Dynamic Programming, Unstructured Grids, Graphical Model.

References

[1] K. Nishida, Y. Ito, K. Nakano. *Accelerating the Dynamic Programming for the Matrix Chain Product on the GPU*. Networking and Computing (ICNC), 2011 Second International Conference on, pp. 320–326, Nov. 30 2011-Dec. 2 2011.

[2] F. Vazquez, G. Ortega, J. J. Fernandez, I.Garcia and E. M. Garzon. *Fast sparse matrix matrix product based on ELLR-T and GPU computing*. Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, pp. 669–674, 10-13 July 2012.

[3] Y. Xia, H. Luo, L. Luo, J. Edwards, J. Lou and F. Mueller. *OpenACC-based GPU Acceleration of a 3-D Unstructured Discontinuous Galerkin Method*. 52nd Aerospace Sciences Meeting. January 2014.

[4] A. di Biagio, A. Barenghi, G. Agosta, G. Pelosi. *Design of a Parallel AES for Graphics Hardware using the CUDA framework*. Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1–8, 23-29 May 2009.

[5] S. Scanzio, S. Cumani, R. Gemello, F. Mana, P. Laface. *Parallel implementation of artificial neural network training*. Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on, pp. 4902–4905, 14-19 March 2010.

[6] N. Bell and M. Garland. 2009. *Implementing sparse matrix-vector multiplication on throughput-oriented processors*. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis , 2009.

[7] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams and K. A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report, EECS Department, University of California, Berkeley, December 2006.

[8] F. Vazquez, G. Ortega, J.J. Fernandez and E.M. Garzon. *Improving the Performance of the Sparse Matrix Vector Product with GPUs*. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, pp. 1146–1151, June 29 2010-July 1 2010.

[9] Fermi Whitepaper: `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidiafermicomputearchitecturewhitepaper.pdf`

Appendices

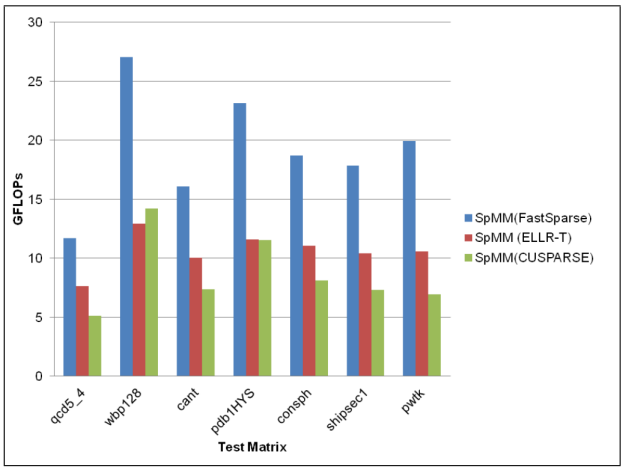


Figure 20: Performances of SpMM on Tesla C2050 [2]

Matrix	N	nz	Av
qcd5_4	49152	1916929	39
wbp128	16384	3933097	240
cant	62451	4007384	64
pdb1HYS	36417	4344766	119
consph	83334	6010481	72
shipsec1	140874	7813404	55
pwtk	217918	11634425	53

Figure 21: NxN Test matrices, nz is the number of non-zeros, Av the average number of non-zeros per row [2]

	Tesla C2050	GTX480
Peak GFlops (single precision)	1030	1350
Peak GFlops (double precision)	515	168
Bandwidth (GB/s)	144	177.4
Clock (GHz)	1.2	1.4
Device memory (GB)	2.6	1.5
Cores	448	480

Figure 22: Characteristics of the GPUs [2]