Applications of Berkeley's Dwarfs on Nvidia GPUs

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

1 The Dwarfs

1.1 Dynamic Programming

The Dynamic Programming Dwarf talks about the Matrix Chain Product problem which can be solved by dynamic programming and how it use 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 1 shows an example of two different parentheses of this matrix chain. Figure 1(a) shows $((A_1 \times A_2 \times A_3 \times A_4) \times (A_5 \times A_6))$, the result of $(A_1 \times A_2 \times A_3 \times A_4)$ is $2 \times 9 \times 3 + 2 \times 3 \times 1 + 2 \times 1 \times 4 = 68$, and the result of $(A_5 \times A_6)$ is $4 \times 11 \times 5 = 220$, so the total number of multiplications is $68 + 220 + 2 \times 4 \times 5 = 328$. Figure 1(b) shows $(A_1 \times (A_2 \times A_3) \times (A_4 \times A_5) \times A_6)$, the result of $(A_2 \times A_3)$ is $9 \times 3 \times 1 = 27$, and the result of $(A_4 \times A_5)$ is $1 \times 4 \times 11 = 44$, so the total number of multiplications is $27 + 44 + 2 \times 9 \times 1 + 2 \times 1 \times 11 + 2 \times 11 \times 5 = 221$.

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

Figure 1: An example of different parentheses of the matrix chain

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 be also clear that $m_{i,i+1} = p_{i-1} \cdot p_i \cdot p_{i+1}$ because $m_{i,i+1}$ means the product of A_i and A_{i+1}, which has no option to choose that is to product these two matrices directly. Generally, to compute the $m_{i,j}$, we should know the solution of its subproblem, that means to compute $m_{i,k}$ and $m_{k+1,j}$ $(i \leq k < j)$ in advance, and then select the best option. So

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

from which $p_{i-1} \cdot p_k \cdot p_j$ 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, use

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

to record the best k. Finally, the whole algorithm is showed as Figure 4. In the line 8, $m_{i,j}$ records the minimum value q and $s_{i,j}$ records the best index k which is helpful to find the relative parentheses.

Figure 2: Matrix Chain Product Algorithm [1]

Figure 3: An examples of table m computed for n = 6 [1]

Figure 5 illustrates an example of talbes m and s for the same matrix chain in Figure 3, 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.

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 4, which computes the costs in the diagonal entries of tables m and s in Figure 5, 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 won't use the current diagonal entries, instead it only uses the values when l is smaller. So calculations of these diagonal entries are independent which can use parallelism. Another way is the loop in lines 6-8 of Figure 4, which computes the value of one entry of tables. It is to search minimum value from subproblems, e.g $m_{1,4}$ is from subproblems $(m_{1,3}+m_{4,4}+p_0 \cdot p_3 \cdot p_4)$,$(m_{1,2}+m_{3,4}+p_0 \cdot p_2 \cdot p_4)$ and $(m_{1,1}, m_{2,4} + p_0 \cdot p_1 \cdot p_4)$. Obviously, each calculation of subproblem can be computed independently and then select the best one. So that it can also be computed in parallel.

Next, Let us consider the amount of the computation for each l. Figure 6 illustrates the number of (i,j) for each l, which shows when l increases the number of (i,j) decreases. And Figure 7 illustrates the number of k for each (i,j) of each l, which shows when l increases the number of k increases too. Also, Figure 8 shows the amount of the computation for each l, which equals to the number of (i,j) and the number of k.

Figure 4: The number of (i,j) for each l [1]

Figure 5: The number of k for each (i,j) of each l [1]

Figure 6: The amount of the computation for each l [1]

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 there are three different types of parallelism OneThreadPerOneEntry, OneBlockPerOneEntry, and BlocksPerOneEntry to solve such problem of various factors.

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 2 is done by one Thread [1]. For example,

$\langle m_{1,4}, m_{2,5}, m_{3,6} \rangle$, each of these entries is computed concurrenty by one core. They use same previous entries $\langle m_{1,1}, m_{2,2}, ..., m_{1,2}... \rangle$ which will be stored in shared memory from global memory in advance.

OneBlockPerOneEntry, this kernel allocates one Block to compute one entry in the tables m and s. The execution in lines 6-8 in Figure 2 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 streaming multiprocessor. So each $(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$ is computed by one core, and another core is to select the minimum one.

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 streaming multiprocessors. So each $(m_{1,k} + m_{k+1,4} + p_0 \cdot p_k \cdot p_4)$ is computed by one core but maybe from different streaming multiprocessors, and some core from some streaming multiprocessor is to select the minimum one.

Evaluation

From Figure 7, 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 is the best choice, and for the third kernel, 128 threads per block and 8 blocks per entry is the best choice. Also, for different range of l (dynamic programming step), the performance of each kernel is different. So the Figure 8 illustrates that the fastest kernel in different range of l so that we can combine these three kernels effectively. Finally, Figure 9 shows the total running time of combinational kernels from GPU and the computing time from 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 actually it is not fair to make comparision although the speed-up factor is about 41 which is a very high value.

Figure 7: Total time of each kernel [1]

Figure 8: Fastest Kernel for different l [1]

Figure 9: Total computing time of GPU and CPU [1]

1.2 Unstructured Grids

The Unstructured Grids Dwarf 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 [9]. Figure 10 is an example that subsonic flow past a sphere with different number of elements, more elements more complex. The number of elements here means the number of 3-d unstructed grids. And this problem can be solved by Discontinuous Galerkin (DG) method. DG method in mathematics form a class of numerical methods for solving differential equations [10]. Also, this method can be implemented in parallel so that it can use OpenACC-based program to accelerate. 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 [11].

Figure 10: An example : Subsonic Flow past a Sphere [3]

Figure 11: Timing measurements for subsonic flow past a sphere [3]

Evaluation

GPU is NVIDIA Tesla K20c GPU containing 2496 multiprocessors which implement the OpenACC-based program ,and CPU is AMD Opteron 6128 CPU containing 16 cores which implement the MPI-based parallel program. Here, Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers [12].

Figure 11 illustrates the different unit running time of GPU and CPU with different number of elements when implementing the simulations of the example showed in Figure 10. Nelem means the number of elements. CPU-1 means using only one core and CPU-16 means using all 16 cores. Obvously, performance of CPU-16 and GPU is much better than one of CPU-1. Also, when number of elements is large enough, GPU has a better performance but not in the first situation. The reason may be 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].

1.3 Graphical Model

The Graphical Model Dwarf talks about Speech Recognition System which uses two different graphic model, one is ANN (Artificial Neural Network) model, another is HMM (Hidden Markov Model). ANN model is for recognizing the acoustic in a time frame (the result is a word or a phoneme). HMM is for warping and adjusting the whole acoustic combining these words or phonemes from ANN model. Here, it focus on using GPU to accelerate training of ANN model.

Figure 12 shows the ANN model, the input is the acoustic in a time frame which is a form of vector, and output is relative word or phoneme. Each node in the graphic represents a value in some vector and the edges represent unknow weights (parameters) to be trained which is used to combine these nodes to get new nodes in next layer. For example, giving the acoustic in a time frame which represent the meaning "dog". But the output of untrained model is the word "cat". So we can use the difference between "dog" and "cat" to adjust the weights of this model. That is training. Also, these weight can be represented by a vector and hidden layer nodes also make up a vector. So, hidden vector equals to the input vector inner products one weight vector ,and output vector equals to the hidden vector inner products another weight vector. Here, "inner product a vector" can be changed to "multiply a matrix". Transform these two weight vectors to relative matrices so that the effect of inner producting the vector is same as one of multiplying the relative matrix.

Because there are input vectors, they can make up a big matrix. Also, the relative output vectors can make up a big matrix. So the input matrix multiplying two weight matrices can get output matrix. For now, 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 problems about linear algebra which can be used for the problem here.

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 [13]. Figure 13 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 here is about memery mapping, with padding can make IO process faster. So we use CUDA with padding comparison with single thread MKL, there is a speed-up factor of 5.

Figure 12: ANN Model [1]

Figure 13: Training time, and relative speed-up for the WSJ0 corpus [1]

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] `http://en.wikipedia.org/wiki/Compressible_flow`

[10] `http://en.wikipedia.org/wiki/Discontinuous_Galerkin_method`

[11] `http://en.wikipedia.org/wiki/OpenACC`

[12] `http://en.wikipedia.org/wiki/Message_Passing_Interface`

[13] `http://en.wikipedia.org/wiki/Math_Kernel_Library`