Gemma in April: A Matrix-like Parallel Programming Architecture on OpenCL

Tianji Wu[*‡], Di Wu[*], Yu Wang[*], Xiaorui Zhang[*], Hong Luo[*], Ningyi XU[†] and Huazhong Yang[*]

[*]Department of Electronic Engineering, TNList, Tsinghua University

Email: wud07@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn

[‡]Department of Electrical Engineering, UCLA

[†]Hardware Computing Group, Microsoft Research Asia

Abstract—**Nowadays, Graphics Processing Unit (GPU), as a kind of massive parallel processor, has been widely used in general purposed computing tasks. Although there have been mature development tools, it is not a trivial task for programmers to write GPU programs. Based on this consideration, we propose a novel parallel computing architecture. The architecture includes a parallel programming model, named Gemma, and a programming framework, named April. Gemma is based on generalized matrix operations, and helps to alleviate the difficulty of describing parallel algorithms. April is a high-level framework that can compile and execute tasks described in Gemma with OpenCL. In particular, April can automatically 1) choose the best parallel algorithm and mapping scheme, and generate OpenCL kernels, 2) schedule Gemma tasks based on execution costs such as data storing and transferring. Our experimental results show that with competitive performance, April considerably reduces the programs' code length compared with OpenCL.**

I. INTRODUCTION

In the battle to achieve higher computing performance, Graphics Processing Unit (GPU) emerges as a novel type of processor for parallel computing. Because of the inherent parallelism of graphics rendering algorithms, which deal with a large number of independent pixels, GPU is a massive parallel architecture with great floating points computing power. These features make modern GPU different from the mainstream general purpose processors in both structure and performance.

The architecture of GPU is suitable for not only graphics rendering algorithms, but also general parallel algorithms in a wide variety of application domains. On the other hand, current high-end GPUs have much more computation power and memory bandwidth than a high-end CPU. Therefore, more and more general-purpose computation applications are mapped to graphics hardware.

However, as a result of its highly specialized parallel architecture, difficulties such as parallel task partitioning, threads communication and synchronization, load balancing, and memory access mode, emerge when implementing GPU-based parallel programs. Designing an efficient GPU program is not easy, and it requires programmers' knowledge of low-level architecture of the underlying chip. Such difficulties extend the development cycle and raise the threshold of GPU parallel programming.

To lower the complexity of parallel programming on GPU while maximizing efficiency, we propose a novel parallel computing architecture, which can help alleviate the difficulties of programming on GPUs. Our architecture includes a parallel programming model, named Gemma, and a corresponding programming framework, named April. Gemma is based on generalized matrix operations, by which programmers can easily transform tradition algorithms

into parallel programs. Then, the framework April can automatically execute Gemma tasks using OpenCL.

This paper is organized as follows. Section II provides our motivations and previous works. In Section III, we present a novel parallel programming model based on general matrix operations - Gemma. In Section IV, we discuss the implementing and optimization of April, our parallel computing framework based on OpenCL. Several application examples with test results and analysis are provided in Section V. Section VI concludes our work and makes the future research plans.

II. MOTIVATION AND RELATED WORKS

A. OpenCL: Open Computing Language

OpenCL is a programming language widely supported since 2009 [1]. It is promoted to unify the GPGPU programming process of AMD and NVIDIA, the programming of multi-core CPU and customized accelerators. That is to say, programs written following the OpenCL specifications can be executed on most mainstream multi-core CPU and GPGPU.

OpenCL is comprised of compilation chain and run-time library. OpenCL compilation chain can compile OpenCL programs statically or dynamically. The run-time library administers the load and execution of programs, initiates the data transmission and so on.

The programming model of OpenCL is intrinsically related to the architecture of GPGPU. For instance, the Work Item in OpenCL corresponds to thread in ATI Stream programming model, the Work Group to thread group and the Local Memory to LDS (Local Data Share) [2], [3].

B. Motivation for General-Purpose GPU Computing Architecture

Although GPGPU is powerful in computing and there is a well-build programming environment, it is still difficult to write effective GPGPU accelerating application programs. Firstly, in order to parallelize their algorithms and map the parallel algorithms to the GPGPU threads, programmers must be familiar with the hardware architecture. Besides, the design of the synchronization of GPU threads and data transmission directly influences the effectiveness of the program. In addition, it is complicated to call runtime library which is designed for high generality. Figure 1 shows the programming flow based on OpenCL.

For the reasons above, a general computing architecture is necessary. A general computing architecture can be divided into three layers. The first layer is the Programming Model, which reduces the difficulty of designing parallel algorithm that needs to be mapped to GPGPU directly. The second layer is the Compiler, which compiles the program with the right Programming Model into the next layer of the framework. The last layer is the Runtime Library, which executes the program and schedules instructions such as data transmission.
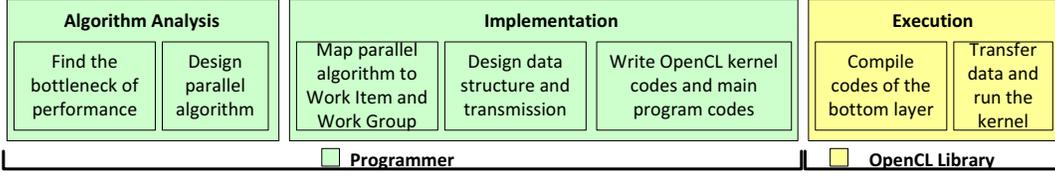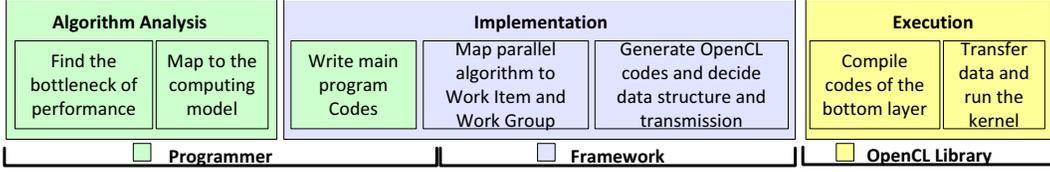
Fig. 1: OpenCL Programming Flow

Fig. 2: Program Design Flow using ideal architecture based on OpenCL.

An excellent architecture could significantly simplify programmers work. Fig.2 shows the workflow of developing a GPGPU program using an architecture under ideal circumstances.

In our novel architecture, Gemma represents the general parallel Programming Model and April consists the next two layers - Compiler and Runtime Library. We design the GPU computing architecture to simplify the programming process of effective high performance GPU programs.

C. Previous work on General-Purpose GPU Architectures

There is a lot of previous work about building up general purpose architectures of different designing goals. For example, MapReduce [4] is a well-known programming model, in which algorithms can be simply represented in two stages, *Map* and *Reduce*, both suitable for parallel systems. Programmers only need to map their algorithms to the model and define the two operations, Map and Reduce. Then MapReduce framework schedules the whole parallel system to complete the algorithm. Drayd [5] is a similar computing model, which generalize the two limited Map and Reduce stages in MapReduce to multiple stages. Drayd uses DAG(Directed Acyclic Graph) to represent operations, in which edges represent transmissions of data.

There are other general computing frameworks for GPU, such as [6]–[8]. Mars [7] has a high compatibility with MapReduce of computer cluster, but targeting at variable-length strings, this framework is not effective with general computing. Catanzaro's architecture [6] is a high performance GPU implementation of MapReduce, but the formats and regularity of data are strictly demanded, thus impair the generality. Tarditi's architecture [8] is based on GPU Pixel Shader, which works effectively but its generality is restricted for the limitation of Pixel Shader.

III. GEMMA: A MATRIX MULTIPLICATION-LIKE PARALLEL PROGRAMMING MODEL

Gemma is a parallel programming model based on generalized matrix multiplication. In this section, we first introduce the basic idea and definition of the model, and then use several examples to illustrate how general algorithms get mapped to the model.

A. Generalization of Matrix Multiplication

Based on the common operations, we can define general matrix multiplication. If two matrices $A_{(U)m \times n}$ and $B_{(V)n \times p}$ and two mappings $\bigotimes : (U, V) \to S$ and $\bigoplus : (S, S) \to S$ follow the rules:

1) $< S, \bigoplus >$ is a monoid, and the identity element is O_S
2) We can always find $O_U \in U, O_V \in V$, which makes $\forall v \in V, O_U \bigotimes v = O_S \ \forall u \in U, u \bigotimes O_V = O_S$

then we define matrix $C_{(S)m \times p}$

$$C_{(S)m \times p} : c_{u,v} = \bigoplus_{i=1}^{n} \left(A_{u,i} \bigotimes B_{i,v} \right) \qquad (1)$$

as product of matrices A and B using $\bigotimes \ and \ \bigoplus$, denoted by $C = (A \times B)_{\bigotimes, \bigoplus}$.

In the definition above, O_U, O_V are called zero elements in U and V. According to the definition, if $U = V = S = \mathbb{R}$ and $\bigotimes, \bigoplus$ are respectively multiplication and addition of real numbers, general matrix multiplication is same with matrix multiplication of real numbers with $O_U = O_V = O_S = 0$.

If we define U, V, S and $\bigotimes, \bigoplus$ following certain rules, the general matrix multiplication above can solve a large number of computing problems. In programming, sets U, V, S can be data structures and $\bigotimes, \bigoplus$ are binary functions. Especially, $\bigoplus$ should follow the Combination Law, because $< S, \bigoplus >$ is a monoid.

The generality of Gemma can be substantiated using the MapReduce Model. By the discussion above, the 'multiplication' of matrix elements can be referred as 'Mapper', and 'addition' of multiplication results can be referred as 'Reducer'. We further extend the compatibility of our model by introducing functional matrix, which is discussed in Section III-C. Therefore, algorithms that can be transformed to the MapReduce structure, can be mapped in the model of Gemma. Furthermore, Gemma supports multiple outputs , since the result of matrix multiplication can be vectors or matrices. This further expands the generality of our model beyond MapReduce.

To illustrate the compatibility of our model, we provide several examples are given in the next subsection to show how to do parallel computing using general matrix multiplication.

B. Application Examples

The first example of generalized matrix multiplication is seeking the maximum element. Given n elements $a_i, (i = 1, 2, .., n), a_i \in \mathbb{R}, (i = 1, 2, .., n)$, we define function $M(x, y)$ as follow:

$$M(x, y) = \begin{cases} x, \text{if } x > y \\ y, \text{otherwise} \end{cases} \qquad (2)$$

Define matrix $A_{(\mathbb{R})1\times n}: a_{1,j} = a_j$ and matrix $B_{(\mathbb{R})n\times 1}: b_{i,1} = 1$. The operation "$\times$" is multiplication of real numbers. Then $c = C_{(\mathbb{R})1\times 1} = (A \times B)_{\times, M}$ is the maximum one of the n elements.

The next example is comparing and swapping. Take a pair of numbers (x_1, x_2) for example. To sort the numbers in a small-to-large order, we construct matrix $C_{\{0,1\}2\times 2}$ as follows:

$$c_{1,1} = c_{2,2} = \begin{cases} 1, & \text{if } x_1 < x_2 \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

$$c_{1,2} = c_{2,1} = \begin{cases} 1, & \text{if } x_1 \geq x_2 \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

Then the product matrix $Y = XC$ is the result of sorting.

C. Further Generalization of Matrix Multiplication

If all n-dimension function with range of S form a set, denoted by $F_{n,S}$, then matrix defined on $F_{n,S}$ is called n-dimension **Function Matrix**. If values of n variables form the set P, then function matrix $A_{(F_{n,S})}$ is the same with a common matrix, denoted by $A_{(S)}(P)$ or $A(P)$, which is a matrix defined on S. If two function matrices $A_{(F_{2,U})m\times n}$ and $B_{(F_{n,V})n\times p}$ and two mappings $\bigotimes: (U,V) \to S$ and $\bigoplus: (S,S) \to S$ follow the rules:

1) $< S, \bigoplus >$ is a monoid, and the identity element is O_S
2) We can always find $O_U \in U, O_V \in V$, which makes $\forall v \in V, O_U \bigotimes v = O_S \ \forall u \in U, u \bigotimes O_V = O_S$

Then we define matrix $C_{(S)m\times p}$ as the product of function matrices A and B using $\bigotimes and \bigoplus$, denoted by $C = (A \times B)_{\bigotimes, \bigoplus}$.

$$C_{(S)m\times p} : c_{u,v} = \bigoplus_{i=1}^{n} \left(A(u,v)_{u,i} \bigotimes B(u,v)_{i,v} \right) \quad (5)$$

In comparison with the definition in Section III-A, here we generalize the common multiplier matrices to function matrices. When calculating value of the element at position u, v of output matrix C, matrices involved in calculating are the value of function matrices at the position u, v. In other words, matrices involved in operation differ according to different objectives calculated.

For instance, suppose we have a matrix of real numbers $D_{(\mathbb{R})2\times n}$. To seek the maximum number in row 1 and the minimum number in row 2 simultaneously, define function column vector $A_{n\times 1}(u,v)$ as follows:

$$a_{i,1}(u,1) = \begin{cases} 1, & \text{if } u = 1 \\ -1, & \text{if } u = 2 \end{cases} \quad (6)$$

The values of j and v are always 1. Using the function $M(x,y)$ defined in III-B, we can get the result immediately: $(c_1, c_2)^T = C = (D \times A)_{\times, M}$. The maximum number of row 1 is c_1 and the minimum of row 2 c_2.

IV. APRIL: AN OPENCL BASED PARALLEL COMPUTING PROGRAMMING FRAMEWORK

April is a computing framework based on Gemma and the OpenCL framework. As discussed in Section II-B, April consists two layers of a general computing framework. The first one is the Compiler, which generates OpenCL codes from the representations of Gemma-based matrix operations (MOPs). The second layer is the Runtime Library, which control the data flow and execution the OpenCL program compiled from the generated codes. April also includes the optimization schemes corresponding to the two layers. In the Compiler layer, April simplifies the programming codes by letting user describing the application into MOPs by Gemma. In the Runtime Library layer, April optimizes the arrangement of each MOP by traversing a Directed Acyclic Graph (DAG) constructed by MOPs. April also selects the best matrix representations for each MOP by comparing their performance models.

Plan is the unit of April programs, which is formed by matrix operations organized by a linear fashion. An operation can be matrix construction (definition), rearranging multiplication and output. The linearity of Plan excludes the branch and loop structures, which need to be designed by users. The workflow of April is showed in Fig.3. The first procedure, Operations Defining, is controlled by users. The rest three procedures, Decision Making, Kernel Codes Producing and Program Executing, can be automatically completed by April.
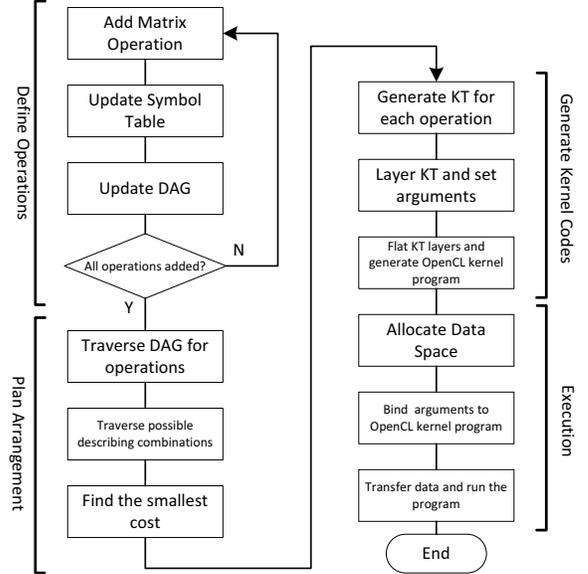

Fig. 3: April Workflow

A. Matrix Representation

The key to April is effective methods of matrix storing. The matrix representation methods are the foundation of April's Runtime-level optimization. In this section, we give four methods to store a matrix.

BUF: The BUF method is for dense matrix storage. In BUF mode, a matrix is stored in a continuous space in the memory. Matrices stored as BUF can be easily accessed but demand a larger storage space. Function matrices cannot be represented as BUF.

XY: In XY mode, we use a function with two arguments x, y [1] to describe a matrix. Stored with the method of XY, matrices can be accessed randomly, and it can represent function matrices. The storage space depends on the function describing the matrix. If the matrix is special and regular, such as diagonal matrix and triangular matrix, then representing it in XY will require much smaller space.

To describe a function matrix using method XY, we need to make a function $\texttt{getXY}$ with input arguments x, y, u, v, which returns the value of element(function) at position x, y of matrix (at point u, v).

CSR: We usually use the CSR (Compressed Sparse Row) method to store sparse matrix [9], which only stores non-zero element in the order of row priority. We need three functions to represent a matrix using CSR:

1) $\texttt{rowsize}(y)$, returning the number of non-zero elements in row $y + 1$.

---

[1]In this article, x is for column and y is for row beginning with 0

2) `col(y, idx)`, returning the column index of the $(idx + 1)$th non-zero element in row $y + 1$.
3) `val(y, idx)`, returning the value of the $(idx + 1)$th non-zero element in row $y + 1$.

y, idx and return value of function `col` all begin with 0.

We can only obtain the value and position of non-zero element of matrix stored with CSR. CSR is better at representing special matrices with higher sparsity. As zero element can be skipped, the speed of sparse matrix multiplication can be greatly improved.

CSC: Similarly with CSR, CSC (Compressed Sparse Column) is a storing method for sparse matrix in the order of column priority.

CSR and CSC are different in the order of storing and accessing non-zero element and thus making CSR matrix better for pre-multiplication and CSC matrix for post-multiplication.

B. Plan Definition

A MOP is constructed by users as an independent object. There are three kinds of MOPs. The first one is the Construction Operation, which has one output matrix and no input matrix. All kinds of data can be bound to the matrices constructed. The second one is the Computing Operation, which has one output matrix and one or more input matrices. The typical example is matrix multiplication with two input matrices and one output matrix. The last one is the Output Operation which has one input matrix and no output matrices. The typical example is writing a BUF matrix to memory or getting the OpenCL Buffer Handle directly.

In all three operations, users can define data structure and interface function with one or more matrix representation methods, such as BUF, XY, CSR and CSC. For general operations, April can support all the possible representation methods. From the supported and user-defined representation methods, April will choose the best one to minimize operation costs.

When a MOP is added to a Plan, a vertex represents that MOP is added to a DAG. In the DAG of operations, every vertices represent MOPs and each directed edge represents the relationship between MOPs or their data flow. For example, an edge from V_1 to V_2 represents that the input matrix of operation V_1 is the output of the operation V_2. The out-degree of each vertex equals the number of input matrices of the operation and the in-degree the number of outputs. Each edge of the DAG indicates the representation method of the destination vertex's output matrix. Different representation methods can result in different performance, which is discussed in Section IV-F. Figure 4 shows an example of a DAG which depicts the construction and computing operations of three matrices, as well as two output operations. By each edge in the graph is the representation method of matrix.

C. Arrangement of MOPs

After the addition of operations, April begins to arrange the execution order and IO matrix representation methods of these operations.

We rearrange the order of operations to get rid of unnecessary operations. The output operation is focused first and then begins the breadth-first post-order traversal to arrange the operations in DAG that depend on the output operation in order. If there are more than one output operations, then start the traversal from them and skip the nodes accessed. In the meantime, the last position accessed of every node is recorded, so that the memory space can be freed when the output data are no longer needed.

As showed in Figure **??**, the breadth-first Postorder Traversal begins from node 6 and we get a sequence:

$$1, 2, 3, 4, 5, 6$$

Then traverse from node 8 and we get:

$$7, 8$$

After linking, we get the final operation order:

$$1, 2, 3(2), 4, 5(3, 4), 6, 7(1, 5), 8(7)$$

The nodes in the brackets are no longer needed($timeoutNodes$).

D. Kernel Codes Generation

With operations in Plan and the matrix representation methods settled, the kernel codes can be generated. An independent kernel program is based on a series of operations with an output matrix represented by BUF. Output matrices represented by the other three methods can only be used as input matrices by the following operation, but they cannot be the output matrices of a kernel program.

To describe the operations of layered structure in Plan, April uses Kernel Template(KT), which is a group of codes containing template variables that can be replaced. We can generate the real codes by replacing the template variables. The template variables can be replaced by other KT, thus forming the layered template structure.

E. Plan Execution

A Plan can be executed after all the kernel codes of matrix operations in the Plan are generated. The execution of a Plan includes data transferring, I/O arguments binding and program executing.

Each operation is executed in the order that rearranged as it is described in Section IV-C. There are three procedures in the Plan Execution, `prepare`, `execute` and `cleanup`, which respectively allocate the output space, execute the operation and free the output space. The allocation and deallocation of input space are accomplished by the parent node.

F. Optimizations

We begin by the performance model of a April Plan. Different matrix representations have the different running cost and output cost. The cost of a node is estimated according to the number of elements accessed in memory when outputting an element, and calculated based on the output method and the output costs of the child nodes.

Assume each operation has a running cost c_r and an output cost c_o, and all the matrices are $n \times n$ square. When the output mode of node p is BUF, the output matrix is calculated by multiplying two dense matrices, and the output matrix is written to the memory instantly. So the output cost $c_o(p) = 1$ and the running cost is

$$c_r(p) = n^3 * \sum_{\text{for each child node } i \text{ of } p} c_o(i) \qquad (7)$$
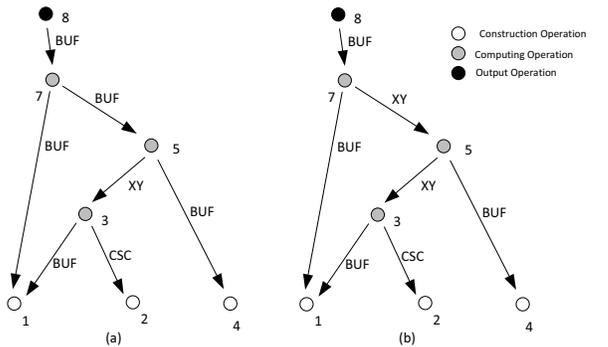

Fig. 4: DAG of Matrix Operations. Different costs of different I/O matrix representation methods

When the output mode of node p is XY, the output matrix is only a function of the input matrices, so the running cost $c_r(p) = 0$ and the output cost is

$$c_o(p) = n * \sum_{\text{for each child node } i \text{ of} p} c_o(i) \tag{8}$$

The performance model of CSR and CSC mode is estimated by the definition of `rowsize`, `col` or `row` and `val` in the same way with BUF and XY. The total Plan running cost is the sum of running costs of all the nodes.

Take the Plans in Fig.4 for an example. Assume that operation 3,5,7 are all matrix multiplications, the computing time of $\otimes, \oplus$ can be ignored, matrices are all square matrix of order n and the sparse matrix constructed by operation 2 is a diagonal matrix.

For plan (a), The total cost is:

$$c_{r(a)} = \sum_{\text{for each node } n} c_r(n) = 5n^3 + 3n^2 + n \tag{9}$$

For Plan (b), the only difference is the output mode of node 5. Therefore, the total cost of the Plan is:

$$c_{r(b)} = \sum_{\text{for each node } n} c_r(n) = 3n^4 + n^3 + 3n^2 + n \tag{10}$$

April traverse all the possible representation methods of I/O matrices, estimate the cost of each combination and choose the most effective one. Therefore, in the example above, April will choose Plan (b) instead of (a).

V. EXPERIMENTAL RESULTS

In this section, we present some of our experimental results to discuss the advantages of our generalized computing architecture: Gemma and April.

A. Experiment Platform and Test Methods

The following experiments are conducted by the same test computer, equipped with AMD PhenomII 965 64-bit 3.4GHz quad-core processor, 8GB DDR3 1333MHz RAM and ATI Radeon HD 5870. The interface between the graphics card and processor is PCI Express x8 with bandwidth of 2.5GB/s. The operating system kernel is Linux 2.6.31 and we use GCC 4.4 as compiler. The driver for GPU is Catalyst 10.5 and the version of OpenCL SDK is ATI Stream SDK 2.1.

All testing programs are executed repeatedly and we calculate the average performance. The executing time of the Plan, including time of data transmission and API, reflects the computing performance of April, but the compiling time is excluded.

B. Improvement of Code Amount

In the section, we discuss code amount differences between pure OpenCL and April. The application is a simple 1024×1024 matrix multiplication program.

$$R = AB \tag{11}$$

A and B are 1024×1024 matrix with single-precision floating-point elements.

Table I shows the code amount of matrix multiplication on different platforms. Compared with pure OpenCL, April significantly reduces the amount of code, because April can automatically finish much work, such as initialization, setting and scheduling. In this case, matrix multiplication is the basic operation of April. Users do not need to write any kernel codes by themselves. In fact, even if it is necessary for users to write some kernel codes, the difficulty of programming can be reduced by April.

TABLE I: 1024×1024 Float Matrix Multiplication Code Amount

Platform	Scheduling Codes	Kernel Codes(Computing Codes)
C	10	10
April	40	34(auto-generated)
OpenCL	1050	96

TABLE II: Comparison I of Different Output Describing Methods

Plan	Time(ms)	Performance(GFLOPs)	Cost
(a)	65.8	32.637	2152726528
(b)	90.3	23.782	4297064448
(c)	163.8	13.110	5370806272

C. Comparison of Different I/O Methods

As discussed in Section IV-F, April can choose for every operation the best output methods among BUF, XY, CSR and CSC. In this section, we discuss the influence of different output methods on computing performance.

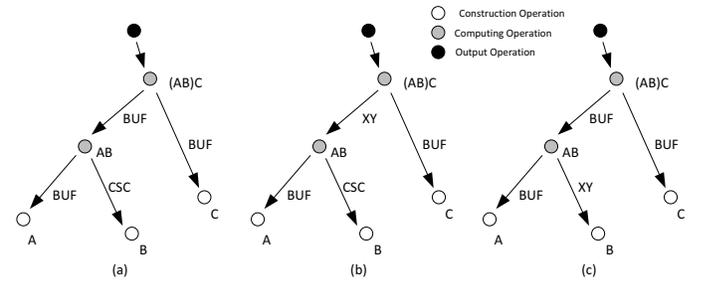Take the computing task of matrix multiplication for example.

Fig. 5: DAG of Different Output Methods

We use DAG to represent the computing task, as showed in Figure 5. There are three different Plans among which Plan (a) is chosen by April. Table II shows the performance and cost estimated by April of three different Plans. We find a big difference among different Plans and April chooses the best one.

Take the computing task showed in Figure 4 for another example:

$$R = M_1 \left((M_1 M_2) M_4 \right) \tag{12}$$

All matrices are 512×512. Matrix M_2 is a diagonal matrix with constant diagonal elements, so $c_o(M_2) = 0$. The cost of the computation is $2(2n^3 - n^2) + n^2 = (4n^3 - n^2)$FLOPs.

Table III shows the result of this experiment. We can see the performance of different Plans are different and April chooses the best one.

D. Full Example: Bitonic Sort

Bitonic Sort [10] is a sorting algorithm of high parallelism. For a group of data $v_i, i = (0, 2, .., 2^b - 1)$ with the length $N = 2^b$, Bitonic Sort can finish the sorting by b steps. Each step s includes s parallel sorting procedures and each procedure includes $N/2$ comparison and swapping. As a result, the complexity of the algorithm is $O(N(\log N)^2)$.

The algorithm of Bitonic Sort can be mapped to the Programming Model of Gemma. Assuming $N = 2$, we can finish the sorting in the

TABLE III: Another Comparison of Different Output Methods

Plan	Time(ms)	Performance(GFLOPs)	Cost
(a)	20.6	26.049	537395200
(b)	3094	0.173	137573695488

order from small to large:

$$\begin{pmatrix} r_0 & r_1 \end{pmatrix} = \begin{pmatrix} v_0 & v_1 \end{pmatrix} X^{(1,1)} \tag{13}$$

The matrix $X^{(1,1)}$ is defined as follow:

$$X^{(1,1)} = \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, & \text{if } v_0 < v_1 \\ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, & \text{otherwise} \end{cases} \tag{14}$$

The best performance is achieved if this swapping matrix is described with CSC. The method CSC needs three functions defined, `colsize`, `row` and `val`. Obviously, there is only one element with value 1 in each column of a swapping matrix. Algorithm 1 shows how to define the three functions of the swapping matrix of Bitonic Sort $X^{(s,p)}$. s and p are current step and procedure, $s = 1, 2, .., b; p = 1, 2, .., s$, as showed in Algorithm 1.

Algorithm 1: Swapping Matrix of Bitonic Sort with CSC: Step s and Procedure p

---

function `colsize`(*x*)
   | Return 1 ;
end
function `row`(*x, idx*)
   | $x' \leftarrow x$ `XOR` 2^{s-p} ;
   | $c_1 \leftarrow (v_x < v_{x'})$;
   | $c_2 \leftarrow (x$ `AND` $2^{s-p} == 0)$;
   | $d \leftarrow (x$ `AND` $2^s == 0)$;
   | **if** c_1 *XOR* c_2 *XOR* d **then** Return x ;
   | **else** Return x' ;
end
function `val`(*x, idx*)
   | Return 1 ;
end

When programming Bitonic Sort using April, we input the original data by row vector and each procedure is a matrix described with CSC. For each Bitonic sort p in each step s, we initialize the sort matrix $X^{(s,p)}$ and add the MOP to the Plan. All the MOPs are added to Plan by loop structure.

Figure 6 shows the performance comparison of GPU Bitonic Sort based on April, quad-core CPU and GPU. The quad-core CPU and GPU implementation is based on OpenCL, and all the input data is floating point. As is illustrated in the figure, when the amount of data is small, April's performance is less than CPU and GPU. This is mainly because 1) it takes time to transmit data between main RAM and VRAM and 2) there are several procedures in Bitonic Sort, each one is a GPU kernel program, so the cost of calling API cannot be ignored. However, when the data scale increases, the performance of April becomes better than CPU. Eventually, April's performance is approximate to optimized GPU implementation [11].

VI. CONCLUSION AND DISCUSSIONS

In this article, we propose a novel parallel computing architecture. The architecture includes Gemma, a general parallel programming model, and April, a programming framework based on Gemma and OpenCL. Gemma uses matrix operation, especially matrix multiplication, to describe general computing tasks. Because the Gemma model is based on matrix, the parallelism is unrelated with the hardware platform. In other words, computing tasks described by
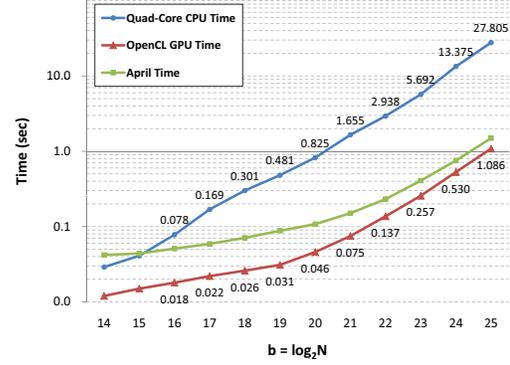

Fig. 6: Performance comparison between quad-core CPU, GPU and April implementation of Bitonic Sort. Blue line with circle label is the performance of quad-core CPU, read line is GPU and green line is April.

Gemma can be applied to any platform including multi-core CPU or GPGPU. April can choose the proper combination of output methods of matrices, so that the cost of the whole plan is optimized.

In the future, we have the following research objectives. Firstly, we will further justifies the generality and compatibility of Gemma by implementing more applications using April. Secondly, the computation can be automatically divided into several parts and controlled by scheduling schemes, especially when the data scale exceeds the Video RAM (VRAM) capacity. Finally, our performance modeling techniques based on DAG can be improved and more detailed. We can further apply the technique to enable April to utilize heterogeneous platforms for task scheduling.

REFERENCES

[1] *The OpenCL Specification*, Khronos OpenCL Working Group, May 2009.
[2] *ATI Compute Abstraction Layer (CAL) Programming Guide*, Advanced Micro Devices, Inc., Mar 2010.
[3] *ATI Stream OpenCL^{TM} Programming Guide*, Advanced Micro Devices, Inc., Jun 2010.
[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.
[6] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.
[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 260–269.
[8] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 325–335.
[9] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM Publication, 2000, ch. Common Issues: Sparse Matrix Storage Formats, pp. 403–404.
[10] K. E. Batcher, "Sorting networks and their applications," in *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*. New York, NY, USA: ACM, 1968, pp. 307–314.
[11] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 325–336.