OpenCL Vector Swizzling Optimization under Global Value Numbering

Li-An Her
Department of Computer Science
National Tsing Hua University
Hsinchu 30013, Taiwan
laher@pllab.cs.nthu.edu.tw

Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30013, Taiwan
jklee@cs.nthu.edu.tw

***Abstract*—Under Heterogeneous System Architecture (HSA), each device work together to improve performance. Open Computing Language (OpenCL) provides functionalities to complete parallel computing in different devices. OpenCL vector bundles identical data types and operates them meanwhile. However, optimization upon vector swizzling becomes difficult due to vector swizzling operations with several OpenCL vectors. In this paper, new method has its announcement to focus on and handle such problem. Vector swizzling optimization takes place under LLVM Global Value Numbering (GVN) optimization. In addition to fundamental functionality of congruence detection in LLVM GVN, new solution in this paper checks relationship between current definition and predecessors and find substitution of current variable with vector swizzling of the predecessors. The result is promising and has many applications on OpenCL optimization and performance breakthrough.**

Keywords—OpenCL; Vector Swizzling; LLVM; GVN;

I. INTRODUCTION

With clock rate in CPU more and more difficult to raise up, modern computation tends to focus on cooperation between each device and component in computing system. Under Heterogeneous System Architecture, computation behaviors are under execution in several devices which architecture may be far different from another.

Implementation of such idea applies Open Computing Language. OpenCL programs are independent from hardware platforms. They can run on many devices such as CPU, GPU, DSP, and even FPGA only when hardware vendors provide their OpenCL libraries and compilers.

OpenCL provides syntax of vectors. Vectors bundle same data types into groups up to 16. In computer graphics, many algorithms and applications use vectors to store and compute 3D objects. For instance, positions, velocities, accelerations of 3D objects are in the form of vectors with three real numbers.

Another application in computer graphics are quaternions. Operations such as rotations complete in quaternions faster than in Euclidean domain. Formation of quaternions are in vectors with four real numbers.

During three-dimension game programming, data structures similar to OpenCL vectors are also available. For instance, the rendering engine OGRE 3D[1] applies the data structure *Vector3* to store and compute positions, velocities. In addition, the data structure *Quaternion* handles relative operations.

In OpenCL, data for positions, directions, velocities, accelerations can be in the form of vectors with three real numbers. The substitution of quaternions in OpenCL is vectors with four real numbers.

Compiler analysis and optimization for OpenCL vectors may be complicated due to vector swizzling. During vector swizzling, vector components have rearrangement, replication, and permutation. For instance, value of variable a is the result of addition operation between vector swizzling of variable b and vector swizzling of variable c. The mathematic formula is below.

$$a.xyzw = b.wzyx + c.xxxy \tag{1}$$

When vector swizzling operations take place much more frequently, the analysis of data dependence between each vector becomes complicated.

One example is a code fragmentation of two arithmetic operations. Two vectors *a* and *b* with four components are available and have their definitions. First operation is the multiplication of swizzling of *a* and swizzling of *b*. New vector *c* with four components keeps result of the multiplication.

Second operations apply assignment, multiplication, and addition operators. The operands are also vector *a*, vector *b*, and a constant value. New vector *d* with four components keeps the result.

Under this circumstance, compiler analysis may not find subtle dependence between each vector. Further optimization is then never under operation. One instance of such case is below [1].

$$\begin{aligned} &c.xyzw = a.xwwz * b.xxxx \\ &d.xyzw = a.wwxz * b.xxxx + 5 \end{aligned} \tag{2}$$

This research is in part supported by Industrial Technology Research Institute (ITRI), 2018. In addition, we would like to thank MOST funding for our research work.

[1] More information about OGRE 3D is available on its official website: https://www.ogre3d.org/

With observation, left operand of addition operation in second formula does have data dependence on first formula. The operand in second formula then can have replacement with vector swizzling of vector *c*. The substitution completes in below formula.

$$\begin{aligned} &c.xyzw = a.xwwz * b.xxxx \\ &d.xyzw = c.yzxw + 5 \end{aligned} \tag{3}$$

Another example shows copy propagation for vector [1]. Two expressions form such case. First expression is just a pure assignment. New vector b with four components keeps result of vector swizzling of vector *a* with four components. Second expression handles addition with vecto swizzling of vector *b* and a constant value. New vector *c* keeps the result. One instance is in below formula.

$$\begin{aligned} &b.xyzw = a.yzwx \\ &c.xyzw = b.zwxy + 5 \end{aligned} \tag{4}$$

In this case, one obvious fact is easy to find that vector wizzling of vector *a* can just replace vector swizzling of vector *b* in second expression. The revision completes as the form below.

$$\begin{aligned} &b.xyzw = a.yzwx \\ &c.xyzw = a.wxyz + 5 \end{aligned} \tag{5}$$

Due to complexity of vector swizzling operations, vector optimizations become complicated during compiler analysis and optimization. Main reason is property of vector data structure. In addition to relationship between variables, the relationship between components in vectors should also be under consideration, which grows the complexity.

During compiler optimization of vector operations, global value numbering is one of the most important optimizations. During GVN, each expression in Static Single Assignment (SSA) form will have one value number. The key of GVN is the detection of congruence. The situation of two expressions with same value number (congruence) means these two expressions have the same result.

The optimization of GVN substitutes the later expression with the earlier one. This reduces redundant or unnecessary expressions and improves performance. GVN also finds congruence of two expressions that is independent from each other but has the same result.

Current compiler project of open source, LLVM[2], does not handle cases of OpenCL vectors above in its new GVN pass. Still, it does not mean GVN fails to finish its job. Actually, LLVM GVN does detect two identical OpenCL vectors. The key is the limitation for congruence of OpenCL vectors is too high.

[2] more information about LLVM: https://llvm.org/
This paper applies LLVM version 5.0.1.

Just like other normal data types, congruence of OpenCL vectors happens only when each corresponding component between vectors is identical. However, the reason why GVN detects congruence is find chance for variable substitution. Now that the desired optimization is replacement, it will be such a pleasure to substitute original operands with vector swizzling of earlier results. Although two variables are not strictly congruent, the goal for optimization does meet.

It will then be obvious to find new algorithms or methods that can bind GVN and vector swizzling optimization into one. In this moment, the expansibility of GVN become important.

GVN optimization is famous for its expansibility. Constant folding, copy propagation, partial redundancy elimination, etc. can run under architecture of GVN. The performance can make further improvement. New methods are available under GVN architecture that also handles vector swizzling optimization

In this paper, new methods and algorithms from the researchers inserts further OpenCL vector optimizations under architecture of Global Value Numbering.

When current expression handles non-vector data types, original GVN works well. When current expression meets OpenCL vector architecture, compiler optimization applies further OpenCL vector optimizations.

Such new design should come to success. The target for optimization is OpenCL vector. Such kinds of data structure are frequently on duty in field of computer graphics and three-dimension game programming.

Applications of modern computer graphics and game programming tend to execute their computations on Graphic Process Unit (GPU). Their environments are under HSA, so OpenCL becomes one of their choice.

New algorithms in this paper removes unnecessary or redundant instructions of OpenCL vectors. This guarantees performance improvement for OpenCL vectors. Then, such breakthrough ensures optimizations on information such as positions, directions, velocities, accelerations, or data structure such as quaternions, which implementation relies on OpenCL vectors.

Instructions of OpenCL vectors are under thorough analysis and careful optimization. Information of three-dimension data and powerful mathematic formula such as quaternions can be under efficient execution. Applications come to a breakthrough. Optimization from researchers in this paper is promising.

II. Related Work

New algorithms and methods in this paper combine traditional Global Value Numbering analysis and advanced OpenCL vector swizzling optimizations.

The applied compiler infrastructure is LLVM. GVN works during intermediate representation (IR). LLVM infrastructure provides convenient way to develop and test IR optimizations. Researchers can directly test their new methods without activation of too many IR passes. They prepare IR code for test, put and generate result IR code from their design, and analyze the behavior. Under LLVM infrastructure, the development and experiments become much easier.

The applied version is LLVM 5.0.1. LLVM also requires ***libclc***[3] to compile OpenCL kernel programs into LLVM IR code.

The original GVN mechanism is from LLVM New GVN. From its comment of implementation file[4], three researches handle the GVN design.

Karthik Gargi introduces new algorithm for global value numbering [2]. He discusses two available algorithms: optimistic and pessimistic algorithms [3, 4, 5]. After analysis of their advantages and disadvantages, he applies his algorithm, which strikes the balance between efficiency of compile-time analysis and abilities to find congruence.

His algorithm assumes only entry basic block is reachable (other basic blocks are unreachable) and each instruction is only congruent to itself. Then, compiler analysis starts to detect actual situations. When process finds one basic block reachable, the table for basic block has to be under modification. When process finds one instruction is also congruent to another one, table for instruction congruence also has to be under revision.

Even though this algorithm works well, other problem appears during execution of Global Value Numbering analysis. When current operation needs results from later operations (back edge happens in control-flow graph), the analysis becomes complicated.

Keith Cooper et al. discusses and finds solutions in their research [6]. They apply two algorithms to solve such back edge problems. First algorithm has to run several times until each value number of instruction stays stable. The time complexity has relationship with multiplication of maximum back edge number and number of definition in source program.

They also come up with second algorithm for compile-time efficiency. They use work from Robert E. Tarjan [7]. Work from Tarjan is to apply depth-first search in order to find strongly connected components (SCC) of graphs in linear-time complexity.

Keith Cooper et al. applies Tarjan's work and data structure of stacks to refine their second algorithm. They find definitions of current SCC has always already been out of stacks. With such order, problems from back edges then disappear.

Another problem during GVN analysis is the φ-function in SSA form. When each operand in φ-function has actually the same value number, then such φ-function can have replacement with another variable. Rekha R. Pai discusses such problem [8]. Her method applies data flow analysis to solve φ-node in polynomial time.

[3] ***libclc*** is an tool for LLVM front-end (clang) to compile OpenCL kernel programs into LLVM IR programs. More information about ***libclc*** are available in its official website: https://libclc.llvm.org/

[4] corresponding source code is available from LLVM github: https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Scalar/NewGVN.cpp

LLVM New Global Value Number optimization handles congruence of Herbrand equivalences [9]. Congruence of two instructions happens when both operators and operands should be the same in correspondence.

Original purpose for Herbrand equivalence is for compile-time efficiency. When such equivalence applies in OpenCL vector, further optimization has no room for application.

Yu-Te Lin et al. discusses further OpenCL vector swizzling optimizations [1]. They detect vector flow dependence, and handle vector aggregation and vector copy propagation. They implement their work under classical architecture of data flow analysis. They state vectors in the form of data access functions to have thorough analysis and advanced optimization [10, 11]. The advantages of their research are the available analysis and optimization on those instructions who are not Herbrand equivalence but can state as vector swizzling form of others.

Then, the goal for researchers in this paper is obvious and clear. New algorithms and methods combine traditional methods and algorithms in LLVM New GVN and advanced OpenCL vector swizzling optimizations in research from Yu-Te Lin et al. [1].

At the sight of LLVM New GVN pass, new design extends and handles advanced OpenCL vector optimizaitons. In addition to traditional strict Herbrand equivalence, new method also finds vector swizzling form of earlier instructions for further substitution and replacement. Such difference has superior advantages from traditional one.

At the sight of OpenCL vector swizzling optimizations in [1], new method applies them into architecture of Global Value Numbering. In addition to OpenCL vector, new design also handles other normal data types. Traditional analysis handles optimization without SSA properties into consideration. New method takes SSA and GVN into consideration.

III. Fundamental

Core concept to form LLVM Global Value Numbering mechanism is the research from Karthik Gargi [2]. When process handles cycles, it applies concept from Keith Cooper et al. [6]. During φ-node, work from Rekha R. Pai handles such case.

Main difference between work of Keith Cooper et al. and actual implementation in LLVM is simplification. Comments on the program indicates original work to find SCC is too powerful and tends to find cycles that are actually find. One example for such cycle for several nodes is that edges result from the same variable. To make it clear, one SCC with two nodes exists. Node A goes down to node B. New definition in node B only uses definition *a* from node A. In node A, another definition *c* needs definition *b* from node B. Then, in such situation, it is fine.

Modification from SCC algorithm in LLVM with Nuutila's improvement applies searching in use-define chain instead of traditional method. When process evaluates whether current target form cycles, it applies modified SCC algorithm and the evaluation is able to complete.

With work from Karthik Gargi, initial assumption for engaged function (or routine in his research) is that only entry basic block is reachable and each definition is only congruent to itself. When process goes through, status of those blocks and definitions may change due to real situation. Blocks, who are not reachable initially, will be reachable after process executes to those blocks and makes sure they are reachable.

With such methods, work form Karthik Gargi can complete several optimizations such as dead block elimination, forward propagation, etc.

When computation graphs have back edges, some parts of those graphs form cycles of same variables. Under such circumstance, modified SCC algorithm finds out those cycles so that process can handle them easier.

Although work from Yu-Te Lin et al. discusses and introduces optimizations on OpenCL vector swizzling [1], their work has yet to take Global Value Numbering into consideration. To run OpenCL vector swizzling optimization under Global Value Numbering, new design for OpenCL vector optimization is necessary.

New algorithm from researchers in this paper also applies data access functions to handle relationships between each component in OpenCL vectors. Based on concepts from [1, 10], The data access functions for OpenCL vectors should contain one variable for data type of OpenCL vector and one index function.

OpenCL vectors bundle several components with same data types. The data structure of OpenCL vectors at least contain base address for those data and the size of vectors. Another index functions help indicate how result looks like in current instructions. For instance, vector swizzling of temporary vector b with four components is *b.xxxx*. The index function in this expression will be *f(i) = 0, where i is an integer from 0 to 3*.

Then, definition of such data access function can state as formula below.

$$\boxed{\begin{array}{l} m' \in \mathbb{N}, \otimes \in \{vector\ operations\}, \\ (\forall\, i \in \mathbb{Z} \cap [0, m'], P_i \in \{OpenCL\ vectors\}), \\ (\forall\, i \in \mathbb{Z} \cap [0, m'], f_i : \mathbb{Z} \to \mathbb{Z}), \\ P_0 f_0 = \otimes_{m=1}^{m'} P_m f_m \end{array}}$$

Fig. 1. Definition of OpenCL vector data access function.

New algorithm is under foundation of data access function. When process handles instructions of OpenCL vectors, it first transforms them into their data access function forms. Then, analysis detects congruence with information on those data access functions.

Detection of congruence in this paper is nearly the same as LLVM New GVN mechanism with one subtle difference. LLVM GVN algorithm finds Herbrand equivalence. Two definitions have Herbrand equivalence with each other when both operator and operands are equivalent.

The problem not to find further optimizations of OpenCL vector is due to property of group of several identical data types for OpenCL vector. With careful vector swizzling, the result operand becomes identical to the other. The optimization can be on its duty.

New method in this paper handles such case with data access functions. In order to meet Herbrand equivalence, new method first checks the equivalence of operators between definitions containing OpenCL vectors. This examination is easy to complete since data access functions keeps information of operations.

Another condition requires size of domain set in one data access function should also meet with the other. Then, two conditions evaluate whether or not current definition has congruence with others: the order of OpenCL vectors in data access functions should be the same with the other, and each values of index functions from each element in domain set should also be in the same order and exist in another data access function.

When current OpenCL vector operations are commutative, compiler analysis can adjust one instruction so that the order may match the other. One example for OpenCL vector congruence evaluation is available below.

$$\begin{aligned} &c.xyzw = a.yyzx + b.wxxy \\ &d.xyzw = a.yxzy + b.wyxx \end{aligned} \tag{6}$$

The operators are the same. The order of vectors are also *a* and *b*. Order for each value from index function in second expression is in the same order and exist in first expression (a.y and b.w, a.x and b.y, etc.). Then, GVN for OpenCL vectors modify the program segmentation as below one.

$$\begin{aligned} &c.xyzw = a.yyzx + b.wxxy \\ &d.xyzw = c.xwzy \end{aligned} \tag{7}$$

This is fundamental congruence evaluation and detection for OpenCL vectors. Such critical mechanism forms and implements whole OpenCL vector optimization under original architecture of LLVM New Global Value Numbering analysis and optimization.

IV. Implementation

The environment for implementation of new methods and algorithms from researchers in this paper is LLVM infrastructure. Based on workflow of LLVM, *clang* of LLVM front-end transforms source programs of OpenCL into IR code. The transformation requires *libclc*.

During analysis of LLVM intermediate representation, researchers' designs of Global Value Numbering with further OpenCL vector analysis and optimization handles the target programs in LLVM IR instructions.

Main designs for Global Value Numbering remain the same as original LLVM New GVN. The obvious difference between LLVM New GVN and researchers' work is the extension for OpenCL vector. From source programs to LLVM IR programs with *libclc*, vector swizzling operations tends to in the form of LLVM *shufflevector* instruction. Researchers focus on analysis and optimizations on *shufflevector* and other vector operations such as additions or multiplications.

Fundamental operation for value numbering is the way to do symbolic evaluation and congruence finding. For symbolic evaluation, congruence class has its leader class. LLVM tends to take those whose ranking number is the lower as leader class. In research from Karthik Gargi, constant numbers, and those numbers who tend to be under early engagement tend to have lower ranking number.

In this paper, steps for congruence finding remain the same as work from Karthik Gargi. The difference from researchers' work to other researchers and implementation is the modification and revision in symbolic evaluation. Symbolic evaluation for OpenCL vector types is quite similar to Karthik Gargi's one with exception for mathematic formations. Expressions are in mathematic form of sum of product in Karthik Gargi's work. In researchers' work, vectors are in mathematic form of data access functions.

This design strikes the balance between vector optimization and reusability from current LLVM GVN implementation. Under formation of data access function, process easily detects and handles vector operations such as vector swizzling.

Additional condition separates OpenCL vectors from other data types. Other normal data types enter into traditional GVN analysis from LLVM.

During congruence evaluation, fundamental mechanism has already been under consideration in last section. Additional design for variable replacement due to GVN requires one more index function for vector swizzling. Like example in formula six and seven, although second expression has room for substitution with first expression, the substitution vector needs vector swizzling operations for adjustment. The index function keeps vector swizzling order.

When vector congruence happens, current variable has substitution with previous one. The adjustment depends on index function discussed above. Then, optimization of GVN on current instruction completes. Total algorithm in the form of pseudocode is available below.

Concept for equivalence on vector swizzling operations in GVN is to check equivalence of vector and contents of index function. First, the number of operands in data access function should be in the same. For instance, vector *a* is addition of vector *b*, vector *c*, and vector *d*, while vector *e* is addition of vector *b* and vector *d*. Number of operands is not the same, so vector *a* is never congruence to vector *b*.

Second, the type of vector operations should be the same between two vectors. Vector *a* is addition of vector *c* and vector *d*, while vector *b* is multiplication of vector *c* and vector *d*. Vector *a* is never congruent to vector *b* since their operators are not the same.

Third, vectors in operands should also be the same and in order between two vectors. Vector *a* is addition of vector *c* and vector *d*, while vector *b* is addition of vector *c* and vector *e*. Vector *a* is never congruent to vector *b*. One thing is that the order of operands can first complete with sort of rank just like work form Karthik Gargi.

Fourth, for each element in domain of index function, each value from it is also in value set from index function. Due to property of data access function of OpenCL vector, some elements in domain of the other data access function exit that the value from these element is exactly that value in first index function.

These four steps form fundamental algorithm to handle congruence finding and symbolic evaluation. The algorithm is available below.

$$
\begin{array}{l}
(m',\ n' \in \mathbb{N}, \otimes \in \{vector\ operations\}, \\
(\forall\, i \in \mathbb{Z} \cap [0, m'], P_i \in \{OpenCL\ vectors\}), \\
(\forall\, i \in \mathbb{Z} \cap [0, n'], Q_i \in \{OpenCL\ vectors\}), \\
(\forall\, i \in \mathbb{Z} \cap [0, m'], f_i : \mathbb{Z} \to \mathbb{Z}), \\
(\forall\, i \in \mathbb{Z} \cap [0, n'], g_i : \mathbb{Z} \to \mathbb{Z}), \\
D\colon (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}, \\
P_0 f_0 = \otimes_{m=1}^{m'} P_m f_m, \\
Q_0 g_0 = \otimes_{n=1}^{n'} Q_n g_n, \\
m' = n', \\
\otimes_{m=1}^{m'} = \otimes_{n=1}^{n'}, \\
\left(\forall\, i \in \mathbb{Z} \cap [0, m'], P_i = Q_j\right), \\
(\forall\, i \in D(g_1), \exists\, j \in D(f_1), \\
\left(g_1(i), g_2(i), \dots, g_{m'}(i)\right) = \left(f_1(j), f_2(j), \dots, f_{m'}(j)\right))) \\
\to \\
(h_0\colon \mathbb{Z} \to \mathbb{Z}, Q_0 g_0 = P_0 h_0)
\end{array}
$$

Fig. 2. Global Value Numbering for OpenCL vectors.

With such method, researchers' work can run under current LLVM GVN. This algorithm plays an important role in vector swizzling optimization under LLVM GVN. When two vectors are under evaluation whether they are congruent in *vector way*, this is the judgement method.

This method is quite different from the original one that handles Herbrand equivalence. Two ways for implementations are available. First way is extension of expression functionality in LLVM GVN so that LLVM expression class is able to apply algorithm from researchers. The other way is division of vector data from other data types during construction of value number table. I. e. Value numbers from vectors and those from other types are in different domain spaces.

V. Experiments and Discussion

After compilation process moves into intermediate representation layer, researchers' algorithm starts to analyze OpenCL kernel programs.

Program segmentation for examples in formula two and three in introduction section is under organization in below figure.

```
int4 * a, *b ;
int4 * c, *d ;
c->xyzw =
  a->xwwz * b->xxxx ;
d->xyzw =
  a->wwxz * b->xxxx
  + 5 ;
```

```
int4 * a, *b ;
int4 * c, *d ;
c->xyzw =
  a->xwwz * b->xxxx ;
d->xyzw =
  c->yzxw + 5 ;
```

Fig. 3. Example code segmentation in OpenCL source program.

Before such program in LLVM IR code is into GVN, this program contains 14 lines. After optimization, the size reduces to 10 lines. Two load instructions for vector *a* and *b*, two *shufflevector* instructions for vector swizzling, and one multiplication instructions vanish. One addition instruction changes its operands from multiplication of swizzling of vector *a* and swizzling of vector *b* to vector *c*. Additional *shufflevector* instruction comes to this program to implement vector swizzling of variable c.

The elimination of two load instructions does improve great performance. Before those test programs are into GVN, they are under optimization of LLVM SROA optimization. Such optimization eliminates unnecessary load and store instructions. In some case, constant folding is also under consideration. Thus, LLVM tries its best to remove all unnecessary load/store.

Removal of multiplication instruction also optimizes the program. Though multiplication becomes much more efficient with process of hardware technology, computation complexity makes multiplication operation more complicated than other operation such as movement or addition.

In this case, the performance has obvious breakthrough with so many eliminations of heavy instructions. This effect is much obvious when engaged data type in those instructions is vector. Load, store, multiplication of vectors mean that those operations execute times of component numbers in those vectors.

Advanced compiler optimizations based on Global Value Numbering such as partial redundancy elimination (PRE) or load instruction elimination deal with such case and gain performance with elimination of unnecessary or redundant instructions in the SSA form although such advanced optimizations have yet to be under implementation in this paper.

In original implementation of LLVM GVN[5], the pass handles normal partial redundancy elimination. In addition, it eliminates partial redundant load instruction. One of further applications based on old GVN is for example GVN hoist optimization[6].

Another factor related to memory operations and bus traffic is the data bandwidth. Data bandwidth highly limits efficiency of bus communication. Unnecessary memory operations occupy limited hardware resource of data bandwidth. The performance decreases dramatically.

Those factors become much more severe when the platform is in Graphic Processor Unit. During execution, many cores in GPU executes identical assembly programs at the same time. When that program contains unnecessary memory operations, the total number of unnecessary memory operations become enormous. The data bandwidth is in its full occupation within short time.

Optimizations on elimination of unnecessary and redundant instructions becomes important on those platforms. Due to natural data complexity of OpenCL vectors, operations on OpenCL vectors are more complicated and takes more hardware resource than normal data types. This fact with heavy operations of memory instructions and complicated computation operations under platform such as GPU forms severe performance problem. Optimizations on OpenCL vectors types become necessary and significant.

The following table shows some experiment results for the programs that mainly execute OpenCL vector operations. The elimination ratio indicates how many instructions researchers' algorithm eliminates comparing original LLVM New GVN. Researchers apply their algorithm particularly on programs with OpenCL vectors since their optimization target is vector rather than other data types.

Below formula shows the way to carry out elimination ratio.

$$ratio = (optimized - original)/\ original \qquad (8)$$

TABLE I. Experimiment Results for OpenCL Vector Operations

Item	Before Optimization		After Optimization		elimination
	Shuffle	***Inst.***	***Shuffle***	***Inst.***	***ratio***
GVN 1	4	21	3	17	-19.05%
copy prop.	4	20	1	17	-15.00%
GVN 2	9	48	2	35	-27.08%

[5] Researchers' design is not under this version of LLVM GVN. The GVN here is the old version one. More information on such version is available here: https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Scalar/GVN.cpp

[6] GVN Hoist completes hoist optimization based on result of value number in GVN pass. However, LLVM GVN Hoist pass around version 5.0.0 has some bugs that will make impact on functionality of original programs. Those who takes interests on GVN hoist should take care about this information. The GVN Hoist optimization pass is available on LLVM website: https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Scalar/GVNHoist.cpp

VI. Future Work

Researchers' algorithm in this paper implements further Global Value Numbering analysis and optimizations on OpenCL vectors. The fundamental mechanism for congruence evaluation and detection depends on data access functions.

In work from Yu-Te Lin et al. [1], they have yet to handle GVN situation. Still, they handle other further analysis and optimization on OpenCL vectors.

One advanced optimization from their work is vector aggregation. During optimization of vector aggregation, several expressions with same order of vectors with subset of index functions are under merge operations. Then, they reunite as one large expression. One example is available in below formula.

$$c.xy = a.yy + b.wx$$
$$c.zw = a.yx + b.wy \tag{8}$$

The optimization for such case is also obvious as below formula.

$$c.xyzw = a.yyyx + b.wxwy \tag{9}$$

Their algorithm for vector aggregation optimization applies data flow analysis. Future work for researchers in this paper is the implementation of vector aggregation of OpenCL programs under architecture of Global Value Numbering.

One of further OpenCL vector optimizations is partial redundancy elimination. Several optimizations of PRE for instance are lazy code motion or variable hoisting. Due to property of vector swizzling operations, some cases may complete PRE case with modification or substitution of vector swizzling of OpenCL vector variables. One of example of such case is available below.

In such case, with careful observation and analysis, variable *b* and variable *d* has subtle relationship. The reusability and replacement for them becomes possible. One of available optimizations applies mechanism of variable hoisting. Process completes code motion out of basic block of branch in source programs. The computation hoists up, and temporary variable keeps the result. When process analyzes variable *d*, it replaces operations with that temporary variable.

```
int a ;
int4 e ;
if ( a == 3 ) {
  int4 * b = e.xyyy + 3 ;
} else {
  int4 * c = e.wzyx ;
}
int4 d = e.yyyx + 3 ;
```

Fig. 4. Example code segmentation for OpenCL PRE vector.

Code segmentation of such optimization is available in the following.

```
int a ;
int4 e ;
int4 t = e.xyyy + 3;
if ( a == 3 ) {
  int4 * b = t ;
} else {
  int4 * c = e.wzyx ;
}
int4 d = t.yzwx ;
```

Fig. 5. Possible optimization on OpenCL PRE vector.

Another optimization is load elimination. Load operation are heavy. Load operations on OpenCL vectors are much heavier. With information from Global Value Numbering, partial redundancy elimination, and additional assistance of data access function, unnecessary and redundant load operations are avoidable.

With foundation of Global Value Numbering and data access functions, further optimizations on OpenCL vectors have more rooms for breakthrough. OpenCL vector swizzling optimizations on GVN has so many advanced optimizations to await implementation.

VII. Conclusion

In this paper, new algorithms from researchers implement OpenCL vector swizzling optimizations under architecture of LLVM New Global Value Numbering. Fundamental formula of OpenCL vectors have be under statement of data access function. Then, congruence evaluation and detection is under implementation based on those data access functions.

Then, whole algorithms from researchers in this paper are under implementation. For clarity, methods for congruence evaluation have temporary replacement with naïve algorithm. In practice, the congruence table applies quadratic hash to probe value numbers of variables.

Methods and algorithms from researchers in this paper have fundamental optimizations of OpenCL vector swizzling under GVN. More advanced optimizations such as vector aggregation, partial redundancy elimination, etc. wait for careful discussion and implementation. With powerful expansibility of Global Value Numbering and assistance of data access functions, those advanced optimizations become a little easier to be under construction.

With more and more applications, which data structures similar to vectors, OpenCL vector swizzling optimizations become more and more important. Fundamental data structure in computer graphics is vector with three real numbers, which stores information in Euclidean coordinate. Powerful mathematic operations of quaternions handle enormous complicated operations such as rotations tend to apply vectors with four real number components.

The optimization on OpenCL vector swizzling is promising. With more and more optimizations such as vector aggregation or partial redundancy elimination, applications based on vectors have high potential to make another great progress.

REFERENCES

[1] Yu-Te Lin, Jenq-Kuen Lee, "Vector Data Flow Analysis for SIMD Optimizations on OpenCL Programs," Concurrency and Computa.: Pract. Exper., vol. 28, pp. 1629-1654, October 2015.

[2] Karthik Gargi, "A Sparse Algorithm for Predicated Global Value Numbering," in Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 45-56, 2002

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting Equality of Variables in Programs," in Proceedings of the 15th ACM Symposium on Principles of Programming Languages, pp. 1-11, January 1988

[4] C. Click, "Combining Analysis, Combining Optimizations," Ph. D. Thesis, Rice University, Febuary 1995

[5] M. N. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches," ACM Transactions on Programming Languages and Systems, vol. 13, issue 2, pp. 181-210, April 1991

[6] Keith Cooper and Taylor Simpson, "SCC-Based Value Numbering," Software Practice and Experience, vol. 27, pp. 701-724, 1995

[7] Robert E. Tarjan, "Depth First Search and Linear Graph Algorithms," SIAM Journal on Computing, vol. 1, issue 2, pp. 146-160, June 1972

[8] Rekha R. Pai, "Detection of Redundant Expressions: A Complete and Polynomial-Time Algorithm in SSA," Asian Symposium on Programming Language and Systems, vol. 9458, pp. 49-65, 2015

[9] Oliver Rüthing, Jens Knoop, and Bernhard Steffen, "Detecting Equalities of Variables: Combining Efficiency with Precision," International Static Analysis Symposium, vol. 1694, pp. 232-247, 1999

[10] Gwan-Hwan Hwang and Jenq-Kuen Lee, "A Function-Composition Approach to Synthesize Fortran-90 Array Operations," Journal of Prallel and Distributed Computing, vol. 54, issue 1, pp. 1-47, 1998

[11] Gwan-Hwan Hwang, Jenq-Kuen Lee, Dz-Ching Ju, "An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs," Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp 112-122, 1995