Unsafe Floating-point to Unsigned Integer Casting Check for GPU Programs [1]

Wei-Fan Chiang [2] , Ganesh Gopalakrishnan, and Zvonimir Rakamarić [3]

School of Computing
University of Utah
Salt Lake City, UT, USA

Abstract

Numerical programs usually include type-casting instructions which convert data among different types. Identifying unsafe type-casting is important for preventing undefined program behaviors which cause serious problems such as security vulnerabilities and result non-reproducibility. While many tools had been proposed for handling sequential programs, to our best knowledge, there isn't a tool geared toward GPUs. In this paper, we propose a static analysis based method which points out all potentially unsafe type-casting instructions in a program. To reduce false alarms (which are commonly raised by static analysis), we employ two techniques, manual hints and pre-defined function contracts, and we empirically show that these techniques are effective in practice. We evaluated our method with artificial programs and samples from CUDA SDK. Our implementation is currently being integrated into a GPU program analysis framework called GKLEE. We plan to integrate dynamic unsafe type-casting checks also in our future work.

Keywords: unsafe type-casting check, numerical program analysis, static analysis, GPU program analysis

1 Introduction

Many numerical programs, including those used for physics simulation [19,17] and image processing [19,10] are accelerated by graphics processing units (GPUs) which support high parallelism. Correctness checking is therefore becoming an important issue for GPU program development. Previous tools have incorporated formal approaches for data race checking in the presence of a large number of threads, hierarchical memory spaces, and thread schedules [14,15,5,7]. One issue that directly affects the integrity of values is unsafe type-casting detection. However this aspect has not received much attention in the past. In this paper, we focus on this problem and offer simple practical solutions based on static typing.

[1] Supported in part by NSF CCF 7298529 and 1346756

[2] Supported in part by DOE Award Number 10034350

[3] Please email Wei-Fan Chiang wfchiang@cs.utah.edu for any questions or comments.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Unintended unsafe casting can cause serious problems, including security vulnerabilities [3,4] or result differences with respect to CPU codes [24]. The central cause of the problems is that under certain circumstances, the casting results become undefined. For example, casting from a negative floating-point number (*e.g.* -1.0) to an unsigned integer (*e.g.* *unsigned int* type in C language) is unsafe/undefined, which means the program is permitted to do literally anything on (or even after) this casting operation. However, detecting unsafe type-casting is difficult because the unsafe scenarios can only be triggered under specific conditions. In the previous example, the condition for triggering the unsafe scenario in the floating-point-to-unsigned-integer (FP2UI) operation is that the argument value needs to be negative.

In this paper, we focus on a specific case of unsafe type-casting which is unsafe FP2UI casting. Specifically, we detect the case of casting a negative floating-point number to an unsigned integer. (Checking other unsafe type-casting scenarios such as overflow can be assisted by other tools such as IOC which will be introduced in §2.) FP2UI operation is widely used in many GPU software and there was a bug on unsafe FP2UI previously reported in an actual GPU parallelized medical software [24]. Unfortunately, to our best knowledge, there doesn't exist a framework for checking FP2UI safety for GPU programs. A dynamic symbolic analysis based GPU program checker, GKLEE, was initially proposed for data race detection [15]. Recently, it had been extended and integrated with many techniques for solving various correctness aspects such as atomicity checking [7] and symmetric thread identification [16]. These extensions suggest that GKLEE is a powerful framework for general GPU program correctness checking. In this paper, we present a static analysis method for detecting unsafe FP2UI casting, and the implementation of this method being integrated into GKLEE. Given that GKLEE is currently focused on handling CUDA GPU programs [20], our implementation is currently limited to handle CUDA.

Our static analysis method tracks possibly negative values (for both integers and floating-point numbers) and checks if any of such values is used in FP2UI casting. Our method is conservative. It means that false warnings of unsafe FP2UI may be raised. We employ two techniques for reducing false alarms: manual hints and pre-defined function contracts. Manual hints allow our method to communicate with external sources of such as users or other analyzers. Pre-defined function contracts allow our method to track possibly negative values in high-level abstraction instead of exploring source code level details. This greatly increases the analysis accuracy. Furthermore, this technique allows our method to handle dynamic linked routines whose source code is not available. The contributions of our work can be summarized as follows:

- We implemented a static analysis of unsafe FP2UI casting detection which is applicable for GPU programs and the implementation is being integrated with a GPU program checking framework called GKLEE.
- We investigated the techniques of reducing false warning that include manual hints and pre-defined function contracts. We show that these techniques are useful in practice through realistic examples from CUDA SDK.

program	=	statement	
statement	=	statement ∘ statement	∘ denotes concatenation
	|	variable = expression ;	
expression	=	expression bop expression	binary operation
	|	(ptype)expression	type-casting
	|	func($\exp_0$, $\exp_1$, ...)	function call
	|	($\exp_{cond}$? $\exp_0$: $\exp_1$)	phi node
	|	variable	
	|	value	
bop	=	+ - * / etc.	binary operator
ptype	=	bool	program's expression type
	|	unsigned int	
	|	int	
	|	float	

Fig. 1. Basic Program Syntax

2 Related Work

Tools for detecting unsafe type-casting for sequential programs had been proposed in many previous contexts. These tools can be classified into three categories: dynamic, static, and dynamic symbolic.

Dynamic tools instrument programs and set up conditionals to check casting arguments at run-time. IOC [9] and BRICK [6] are the examples of the dynamic approach. These tools generate no false alarms; however, the detection coverage depends on the testing inputs provided by the users.

Static analysis tools perform type inference. IntScope [23] and our analyzer are the examples of this kind. In contrast to dynamic approaches, static methods guarantee coverage but false alarms may be raised.

Dynamic symbolic analysis is similar to white-box fuzz testing [11] which is driven by some initial inputs. These tools automatically generate new inputs for increased test coverage. SmartFuzz [18] belongs to this category. The test coverage and the scalability of these tools may be limited by the underlying constraint (SMT) solvers.

3 Methodology

3.1 Conservative Tracking of Non-negative Values

The key idea of our static analysis is to track non-negative values and check if there is any type-casting expression taking a possibly negative value as argument.

Figure 1 shows the core syntax of the programs handled by our static analysis. In this syntax, we omit many actual data types in real CUDA (or C) programs such as 8-bit integer (*char* in C language) and 64-bit floating-point number (*double* in C) but keep only one type for each category to simplify our illustration. The preserved program types (*ptype*) are *bool*, *int*, *unsigned int*, and *float* and these are the types that programmers are allowed to declare in their programs. For simplicity, we also omit the relation operations such as *less-than* (<) and *greater-than* (>) in the syntax.

In order to track non-negative values, our analyzer expands the program type

type	=	ptype	the program type specified in Figure 1
	|	non-negative int	for integers whose value are non-negative
	|	non-negative float	for floating-point numbers whose values are non-negative

Fig. 2. Expanded Type used by Our Unsafe Type-casting Check

(1) basic binary operation: $$\frac{exp_0 : \mathrm{T} \;\; bop \;\; exp_1 : \mathrm{T}. \;\; bop \in \{+, *, /\}}{exp_{rel} : \mathrm{T}}$$

(2) integer subtraction: $$\frac{exp_0 : \mathrm{T} \;\; - \;\; exp_1 : \mathrm{T}. \;\; \mathrm{T} \in \{\text{unsigned int}, \text{non-negative int}, \text{int}\}}{exp_{rel} : \text{int}}$$

(3) floating-point subtraction: $$\frac{exp_0 : \mathrm{T} \;\; - \;\; exp_1 : \mathrm{T}. \;\; \mathrm{T} \in \{\text{non-negative float}, \text{float}\}}{exp_{rel} : \text{float}}$$

(4) integer inference of binary operation: $$\frac{exp_0 : \text{non-negative int} \;\; bop \;\; exp_1 : \text{int}}{exp_{rel} : \text{int}}$$

(5) floating-point inference of binary operation: $$\frac{exp_0 : \text{non-negative float} \;\; bop \;\; exp_1 : \text{float}}{exp_{rel} : \text{float}}$$

(6) basic Phi node: $$\frac{(exp_{cond} : \text{bool} \; ? \; exp_0 : \mathrm{T} \;\; : \;\; exp_1 : \mathrm{T})}{exp_{rel} : \mathrm{T}}$$

(7) integer inference of Phi node: $$\frac{(exp_{cond} : \text{bool} \; ? \; exp_0 : \text{non-negative int} \;\; : \;\; exp_1 : \text{int})}{exp_{rel} : \text{int}}$$

(8) floating-point inference of Phi node: $$\frac{(exp_{cond} : \text{bool} \; ? \; exp_0 : \text{non-negative float} \;\; : \;\; exp_1 : \text{float})}{exp_{rel} : \text{float}}$$

Fig. 3. Operational Semantics of Binary Operations and Phi Node Being Integrated into GKLEE

(*ptype*) in background. Figure 2 shows the types used in our analysis. Specifically, we add two additional types, *non-negative int* and *non-negative float*, on top of *ptype*. Type *non-negative int* (or *non-negative float*) is for integers (or floating-point numbers) which are declared as *int* (or *float*) but their values are non-negative. On the other hand, type *int* and *float* (in Figure 2) are for variables whose values **could be** negative. Non-negative values (including *non-negative int* type and *non-negative float* type values) are created by constant assignments. Since our method is static analysis based, some non-negative values may be conservatively inferred as *int* or *float*. We will introduce some techniques for reducing conservative inferences later in §3.2.

We use notation $expr : \mathrm{T}$ to denote an expression *expr* whose type is T. In Figure 3, Rules 1 to 5 show the semantics of binary operation evaluation and Rules 6 to 8 show the semantics of Phi node evaluation. All the semantic rules are symmetric: a rule can be applied to the case of exchanging the two operands' types. Rules 4, 5, 7, and 8 show the conservative type inference performed in our static analysis.

Figure 4 shows the semantics of type casting. The principle rule of casting is that casting a non-negative value to a possibly negative type (*int* or *float*) would results in a non-negative type (*non-negative in* or *non-negative float*). Expression *Warning* in Rule 15 and 20 denotes potential unsafe type casting. Specifically, Rule 20 describes the unsafe FP2UI scenario we focus in this work. A warning message of unsafe type casting will be given by our analyzer if one of these rules is triggered, and the program may have undefined behavior.

(9) basic type casting: $\frac{(\mathrm{T})exp_{op} : \mathrm{T}}{exp_{rel} : \mathrm{T}}$

(10) *unsigned int* to *int*: $\frac{(\mathrm{int})exp_{op} : \text{unsigned int}}{exp_{rel} : \text{non-negative int}}$

(11) *unsigned int* to *floating-point*: $\frac{(\mathrm{float})exp_{op} : \text{unsigned int}}{exp_{rel} : \text{non-negative float}}$

(12) *non-negative int* to *unsigned int*: $\frac{(\text{unsigned int})exp_{op} : \text{non-negative int}}{exp_{rel} : \text{unsigned int}}$

(13) *non-negative int* to *int*: $\frac{(\mathrm{int})exp_{op} : \text{non-negative int}}{exp_{rel} : \text{non-negative int}}$

(14) *non-negative int* to *floating-point*: $\frac{(\mathrm{float})exp_{op} : \text{non-negative int}}{exp_{rel} : \text{non-negative float}}$

(15) *int* to *unsigned int*: $\frac{(\text{unsigned int})exp_{op} : \text{int}}{Warning}$

(16) *int* to *floating-point*: $\frac{(\mathrm{float})exp_{op} : \text{int}}{exp_{rel} : \text{float}}$

(17) *non-negative float* to *unsigned int*: $\frac{(\text{unsigned int})exp_{op} : \text{non-negative float}}{exp_{rel} : \text{unsigned int}}$

(18) *non-negative float* to *int*: $\frac{(\mathrm{int})exp_{op} : \text{non-negative float}}{exp_{rel} : \text{non-negative int}}$

(19) *non-negative float* to *float*– Safe FP2UI: $\frac{(\mathrm{float})exp_{op} : \text{non-negative float}}{exp_{rel} : \text{non-negative float}}$

(20) *float* to *unsigned int* – Unsafe FP2UI: $\frac{(\text{unsigned int})exp_{op} : \text{float}}{Warning}$

(21) *float* to *int*: $\frac{(\mathrm{int})exp_{op} : \text{float}}{exp_{rel} : \text{int}}$

Fig. 4. Type-casting Semantics

3.2 Optimizations

Our static analysis follows the semantics (Figure 3 and 4) that conservatively infer values' types. Without applying any optimization, our analyzer tends to raise many false alarms: giving warnings on safe type-castings. Here we describe some optimizations applied in our analyzer, and we will show how false alarms are eliminated through examples in §4.

Manual Hints Users' knowledge is usually useful for a static analyzer to avoid false alarms. Our current implementation provides an interface for programmers to manually claim non-negative expressions.

Pre-defined Function Contracts Our type-casting check is currently implemented to perform function-wise analysis. Type information is not passed among functions. In other words, for handling an arbitrary function call expression, we simply check the return type described in the function declaration. For example,

(22) absolute value: $$\frac{abs(exp : \mathrm{T}).\ \mathrm{T} \in \{non-negative float, float\}}{exp_{rel} : \text{non-negative float}}$$

(23) vector length: $$\frac{length(exp_0 : \mathrm{T},\ exp_1 : \mathrm{T}, \ldots).\ \mathrm{T} \in \{\text{non-negative float}, \text{float}\}}{exp_{rel} : \text{non-negative float}}$$

(24) ceiling value: $$\frac{ceil(exp : \text{non-negative float})}{exp_{rel} : \text{non-negative float}}$$

(25) floor value: $$\frac{floor(exp : \text{non-negative float})}{exp_{rel} : \text{non-negative float}}$$

(26) taking maximum: $$\frac{max(exp_0 : \mathrm{T}_0,\ exp_1 : \mathrm{T}_1).\ \mathrm{T}_0 = \text{non-negative float} \vee \mathrm{T}_1 = \text{non-negative float}}{exp_{rel} : \text{non-negative float}}$$

(27) taking minimum: $$\frac{max(exp_0 : \text{non-negative float},\ exp_1 : \text{non-negative float})}{exp_{rel} : \text{non-negative float}}$$

Fig. 5. Optimizations with Pre-defined Function Contracts

let function *foo*'s signature is

$$float \rightarrow float.$$

Function call expression $foo(1.0)$ will be considered as possibly negative even though the actual implementation of foo may be

$$float \text{ foo } (float \text{ f}) \{ \text{ return f; } \}$$

that expression $foo(1.0)$ is actually non-negative.

We found that taking pre-defined function contracts such as the mathematical axioms of some math routines into the consideration of type-casting check can greatly reduce false alarms. For example, many programming frameworks (such as CUDA) provide function *abs* which takes a floating-point value as parameter and returns the absolute value. Obviously, the returned value of *abs* is always non-negative. Figure 5 shows the semantics of inferring *non-negative float* type for some function call expressions. Function call evaluations not listed in Figure 5 are just referred to the function declarations.

3.3 Current Limitations

At this point, we only have a prototype implementation of our static analysis method. As we will see an example in §4.2, there are many CUDA (or C language) syntax such as structure and pointer array that our current implementation cannot handle. Cross-function analysis is also not realized in our current implementation that type inference information cannot be passed from a function caller to a callee. We will refine the engineering of our implementation in the future work.

For handling branch statement, we currently consider every path as feasible and, at a join location, conservatively infer types according to all incoming paths. The semantics for handling branches are similar to Rule 6, 7, and 8 which are the semantics of handling Phi node. In our future work, we plan to invoke GKLEE's facility of identifying infeasible path that would increase the accuracy of our unsafe type-casting analysis. For handling loop, we simply performance unrolling. Integration with fixed-point theory [12] is also in our plan.

4 Experimental Results

We collected some examples for evaluating our unsafe FP2UI detection. In §4.1, we demonstrate that our method can successfully report unsafe FP2UI casting in some artificial benchmarks created by us. In §4.2, we demonstrate that our method can mostly avoid reporting false alarms in practice. The examples used in §4.2 were extracted from CUDA SDK (6.0).

The programs for all our experiments (in both §4.1 and §4.2) were compiled with Clang [1] with optimization flag *-O0*. Compiling CUDA programs by Clang is supported by GKLEE's facility of transforming CUDA program to C. Our experiments were performed on a machine with 12 Intel Xeon 2.40GHz CPUs and 48GB RAM.

4.1 Demonstration of Unsafe Type-casting Detection

Figure 6 shows two artificial (CUDA) functions which demonstrate unsafe FP2UI usage. In Figure 6a, the assertion on line 7 ensures that all the floating-point values from the argument array *data* are non-negative. This information (described by the assertion) is passed to our static analysis as a manual hint. In our experiment on the program in Figure 6a with line 8 (with line 9 commented out), our static analysis raised an unsafe type-casting alarm on line 8. The reason is that $(fp0 + fp1)$ is considered as non-negative while $(fp2 - fp3)$ is considered as possibly negative (by Rule 3). Thus, the evaluation result of the Phi node

$$(myid < 256\ ?\ (fp0 + fp1)\ :\ (fp2 - fp3))$$

is possibly negative (by Rule 8). If exchanging line 8 with line 9, the both incoming expressions are non-negative. Thus, no alarm was raised by our analyzer.

Figure 6b shows an example of calling user-defined function (*fAdd*) and math routines (*abs* and *ceil*) that the math routines have pre-defined contracts as shown in Fig. 5. In our experiment on the program in Figure 6b with line 8 (with line 9 commented out), an alarm was raised for the FP2UI on line 10. The reason is that function *fAdd*'s return type is declared as *float* (line 1). Therefore, *fp2*'s value was inferred as possibly negative. Consequently, the FP2UI on line 10 could cast from a negative value. If exchanging line 8 with line 9, Rule 22 is triggered with the call of *abs* (which calculates absolute value). Therefore, *fp2*'s value was inferred as non-negative and the FP2UI on line 10 always cast from a non-negative value (by Rule 24).

4.2 Case Studies on CUDA SDK Samples

We studied some usages of FP2UI casting in CUDA SDK and categorized the examples into two scenarios: thread group size computation and information compaction. We manually analyzed all the usages of FP2UI casting and found that all of them are safe. Our analyzer raised only one alarm (a false alarm) in all our examples from CUDA SDK. In many examples, we observed that simple manual hints and pre-defined contracts help our analyzer avoid raising false alarm.

Computing Thread Group Size Figure 7 shows the extracted code from programs *simpleZeroCopy*, *FDTD3d*, and *cdpBezierTessellation*. Figure 7a shows the usage of FP2UI casting (line 5) in *simpleZeroCopy* for deciding the number of thread

```
1: procedure __GLOBAL__ FOO(float data, unsigned int results)
2:     int myid = threadIdx.x;
3:     float fp0 = data[myid * 4 + 0];
4:     float fp1 = data[myid * 4 + 1];
5:     float fp2 = data[myid * 4 + 2];
6:     float fp3 = data[myid * 4 + 3];
7:     assert(0 ≤ fp0, fp1, fp2, fp3);
8:     results[myid] = (unsigned int) ( myid < 256 ? (fp0 + fp1) : (fp2 - fp3) );
9:     // results[myid] = (unsigned int) ( myid < 256 ? (fp0 + fp1) : (fp2 + fp3) );
10: end procedure
```

(a) Basic Binary Operations and Phi Node

```
1: procedure __DEVICE__ float FADD(float arg0, float arg1)
2:     return arg0 + arg1;
3: end procedure
4: procedure __GLOBAL__ BAR(float data, unsigned int results)
5:     int myid = threadIdx.x;
6:     float fp0 = data[myid];
7:     float fp1 = data[myid + blockDim.x];
8:     float fp2 = fAdd(fp0, fp1);
9:     // float fp2 = abs(fAdd(fp0 + fp1));
10:    results[myid] = (unsigned int) ceil(fp2);
11: end procedure
```

(b) Function Calls

Fig. 6. Artificial Examples for Demonstrating Unsafe Type-casting Detection

blocks in a grid. Our analyzer claimed this usage as safe because it propagated the non-negative constants from *nelem* (1048576) and *block.x* (256), and found that (by Rule 1) the floating-point number

$$(float)nelem/(float)block.x$$

given to function *ceil* is non-negative (by Rule 1). Thus, the floating-point value taken by the FP2UI casting at line 5 is non-negative (by Rule 24).

Figure 7b shows the usage of FP2UI casting (line 9 and 10) for deciding grid dimensions in *FDTD3d*. Function call *checkCmd()* checks if there is any user-specified block size given through command line and function call *getUserBS()* returns the user-specified block size. Our analyzer successfully inferred that integer *userBlockSize* is non-negative (line 4). It firstly decided that both *max* and *min* must return a non-negative integer (by Rule 26 and 27) and then decided the Phi node on line 4 is selecting between an non-negative integer and constant 512. Therefore, both SI2UI (signed integer to unsigned integer) castings on line 5 and 6 are safe. With the manual hints on line 1 that assert both *dimx* and *dimy* are non-negative, our analyzer claimed that the usages of FP2UI casting (on line 8 and 9) in *FDTD3d* are safe.

There are two sections of code in program *cdpBezierTessellation* use FP2UI casting. These sections are shown in Figure 7c and 7d. Figure 7c defines structure *BezierLine* and the constants used in both code sections. The usage of FP2UI casting shown in Figure 7c is similar to the case of Figure 7a (constant propagation). For the usage in Figure 7d, our analyzer successfully inferred that *curvature* at line 4 must be a non-negative floating-point value (by Rule 23) and *nTessPoints* at line 5 must be a non-negative integer. However, our analyzer currently cannot infer that the grid size (*dGrid*) must be created by casting from an non-negative floating-point number because our current implementation doesn't track non-negative value for structure members. In this case, the structure member needed to be tracked is *bLines[lidx].nVertices*. Also, the value of the targeted

```
1: procedure MAIN( )
2:     int nelem = 1048576;
3:     unsigned int bytes = nelem * sizeof(float);
4:     dim3 block = {1, 1, 256};
5:     dim3 grid = {1, 1, ((unsigned int) ceil( (float)nelem / (float)block.x))};
6: end procedure
```

(a) simpleZeroCopy

```
1: procedure FDTDGPU(int dimx, int dimy)
2:     assert(0 ≤ dimx, dimy);
3:     dim3 dimBlock, dimGrid;
4:     int userBlockSize = (checkCmd() ? min(max((getUserBS() / 1024), 128) : 512));
5:     unsigned int dbx = 32;
6:     unsigned int dby = (((userBlockSize / 32) < 16) ? (userBlockSize / 32) : 16);
7:     dimBlock.x = dbx;
8:     dimBlock.y = dby;
9:     dimGrid.x = (unsigned int) ceil((float)dimx / dbx);
10:    dimGrid.y = (unsigned int) ceil((float)dimy / dby);
11: end procedure
```

(b) FDTD3d

```
1: #define N_LINES 256
2: #define BLOCK_DIM 64
3: struct BezierLine {
4:    float2 CP[3];
5:    float2 * vertexPos;
6:    int nVertices; }
7: procedure MAIN( )
8:     BezierLine * bLines;
9:     // some initialization of bLines here...
10:    unsigned int dGrid = (unsigned int) ceil((float)N_LINES / (float)BLOCK_DIM);
11:    computeBezierLinesCDP<<<dGrid, BLOCK_DIM >>>(bLines, N_LINES);
12: end procedure
```

(c) cdpBezierTessellation : main

```
1: procedure __GLOBAL__ COMPUTEBEZIERLINESCDP(BezierLine * bLines, int nLines)
2:     int lidx = threadIdx.x + blockDim.x * blockIdx.x;
3:     BezierLine bl = bLines[lidx];
4:     float curvature = length( bl.CP[1] - 0.5 * (bl.CP[0] + bl.CP[2]) ) / length( bl.CP[2] - bl.CP[0] );
5:     int nTessPoints = min(max((int)(curvature * 16.0), 4), 32);
6:     if bl.vertexPos == NULL then bLines[lidx].nVertices = nTessPoints;
7:     unsigned int dGrid = (unsigned int) ceil((float)bl.nVertices / 32.0);
8:     computeBezierLinePositions<<<dGrid, 32 >>>(lids, bLines, bl);
9: end procedure
```

(d) cdpBezierTessellation : computeBezierLinesCDP

Fig. 7. (Extracted) Code in CUDA SDK Samples for Computing Thread Group Size

```
1: procedure __DEVICE__ unsigned int RGBAFLOATTOINT(float4 rgba)
2:     float sx = saturate(rgba.x);
3:     float sy = saturate(rgba.y);
4:     float sz = saturate(rgba.z);
5:     float sw = saturate(rgba.w);
6:     unsigned int infow = ((unsigned int)(sw * 255.0f) << 24);
7:     unsigned int infoz = ((unsigned int)(sz * 255.0f) << 16);
8:     unsigned int infoy = ((unsigned int)(sy * 255.0f) << 8);
9:     unsigned int infox = ((unsigned int)(sx * 255.0f));
10:    return infow | infoz | infoy | infox;
11: end procedure
```

Fig. 8. (Extracted) Code in CUDA SDK Samples for Information Compaction

structure member (*bLines[lidx].nVertices*) is undefined in the else-path of the if-statement (line 6) of this function. To resolve this issue, cross-function analysis is needed which is currently not in our implementation. We plan to revise our implementation for handling more C (or CUDA) syntax such as structure and handling cross-function analysis in our future work.

Information Compaction Figure 8 shows the code used in programs *boxFilter* and *bilateralFilter* for compacting color information stored in *float4* to an unsigned

Category	Benchmark	Unsafe FP2UI Found by Manual Analysis	Unsafe FP2UI Found w/o Opt.	Unsafe FP2UI Found with Opt.	Applied Optimizations
Artificial	binary op. & Phi node	Y	Y	Y	manual hints
	function calls	Y	Y	Y	pre-def. contracts
CUDA SDK	simpleZeroCopy	N	Y	N	pre-def. contracts
	FDTD3d	N	Y	N	manual hints & pre-def. contracts
	cdpBezierTessellation	N	Y	N	pre-def. contracts
	info. compaction	N	Y	Y	pre-def. contracts

Table 1
Summary of our Experimental Results
Column *Unsafe FP2UI Found by Manual Analysis* indicates whether a benchmark contains any unsafe FP2UI casting. Column *Unsafe FP2UI Found with/without Opt.* indicates whether an unsafe FP2UI warning was raised by our static analysis with/without optimization techniques. *Y* (*N*) indicates that an unsafe FP2UI was found (not found). Column *Applied Optimizations* indicates the optimization techniques applied for the result shown in Column *Unsafe FP2UI Found with Opt.*.

integer. Function *saturate* clamps a floating-point value to the range $[0.0, 1.0]$ which can be interpreted as

$$saturate(fp) = max(0.0, min(1.0, fp))$$

By Rule 26 and 27, *sw*, *sx*, *sy*, and *sz* on line 2 to 5 are all non-negative floating-point values. Therefore, the type casting on line 6 to 9 are all safe.

Table 1 summarizes all our experimental results. Column *Unsafe FP2UI Found by Manual Analysis* shows the results of manual unsafe FP2UI casting analysis. A benchmark is marked with label *Y* (*N*) if it contains (doesn't contain) an unsafe FP2UI casting. Column *Unsafe FP2UI Found without Opt.* shows the results of our static analysis without any optimization techniques applied. A benchmark is marked with label *Y* (*N*) if our analyzer detected (didn't detect) a potentially unsafe FP2UI casting in the benchmark. Column *Unsafe FP2UI Found with Opt.* shows the results of our static analysis with some optimization techniques applied. The applied optimization techniques are specified in the last columns of Table 1. Column *Applied Optimizations* shows the optimization techniques applied for the results shown in Column *Unsafe FP2UI Found with Opt.*.

4.3 Potential Unsafe FP2UI in CUDPP

CUDPP [2] is a GPU computation library which provides many parallel algorithm primitives such as prefix-sum [13,21], sorting [8], and random number generation [22]. We used our static analysis to check the latest version (2.2) of CUDPP and found a potential unsafe FP2UI in it. The potential unsafe FP2UI was reported to the CUDPP developers but our finding is yet not confirmed by the submission deadline of this paper. (Thus, we claim our finding as a potential unsafe type-casting scenario.) Figure 9a shows the extracted code of the function (*FF*) which contains the potential unsafe FP2UI casting in file *ran_gold.cpp* of CUDPP. When the value of variable *t* (in Figure 9a) is negative, the result of the FP2UI in line 3 is undefined. In fact, the code could suffer from another unsafe type-casting scenario when *t* is greater than the maximum limit of *unsigned int* value. (This type of unsafe type-casting is out of the focus of this paper.) However, we found another file, *rand_cta.cuh*, in CUDPP which defines the similar functions as those in

```
1: procedure void FF(uint4 * td, int i, uint4 * Fr, float p, unsigned int * data)
2:     float t = sin((float)(i)) * p;
3:     unsigned int trigFunc = (unsigned int)t;
4: end procedure
```

(a) Potential Unsafe FP2UI in CUDPP (rand_gold.cpp)

```
1: procedure __DEVICE__ void FF(uint4 * td, int i, uint4 * Fr, float p, unsigned int * data)
2:     float t = sin(__int_as_float(i)) * p;
3:     unsigned int trigFunc = __float2uint_rd(t);
4: end procedure
```

(b) Safe FP2UI in CUDPP (rand_cta.cuh)

Fig. 9. The Potentially Unsafe and the Safe FP2UI Type-castings in CUDPP

rand_gold.cpp including the (potentially) problematic function shown in Figure 9a. Figure 9b shows the safe type-casting version (defined in *rand_cta.cuh*) of the function in Figure 9a. In Figure 9b, the type-castings are guarded by CUDA library routines, *__int_as_float* and *__float2uint_rd*, that use the safe type-casting approach suggested in the previous context [24].

4.4 Discussions: Potential Integration with Dynamic Analysis

Our static analyzer is currently integrated with GKLEE. As mentioned in §3.3, we plan to invoke more GKLEE's facilities such as path feasibility check in our future work to improve the detection accuracy (to reduce false alarms). Here we propose a potential integration with IOC [9]: a dynamic unsafe type-casting checker. IOC can check unsafe type-casting scenarios such as overflow which are not handled by our currently static analysis method. Also, IOC inserts code for every type-casting instruction that the instrumented program generates warning if any unsafe casting is triggered in run-time. In other words, IOC not only dynamically detects unsafe casting but also synthesizes safe programs which don't allow silent undefined behaviors. However, the instrumentation causes performance overhead. Our static analyzer can select potentially unsafe type-casting instructions for IOC to insert safety check code. Comparing to using IOC alone, this combination should synthesize programs with higher performance because less safety check code been inserted.

5 Concluding Remarks

Detecting unsafe type-casting is important for GPU program development. In this paper, we presented a static analysis based method for detecting unsafe type-casting and a prototype integration with a GPU program analyzer. We empirically showed that our method can avoid reporting many false alarms in practice. We plan to invoke symbolic analysis to increase detection accuracy and integrate with program synthesis to generate low-overhead safe programs in our future work.

References

[1] *Clang: a C language family frontend for llvm*, http://clang.llvm.org/.

[2] *CUDPP*, http://cudpp.github.io/.

[3] *Cve-2002-0639: Integer overflow in sshd in openssh* (2002), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639.

[4] *Cve-2010-2753: Integer overflow in mozilla firefox, thunderbird and seamonkey*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753 (2010).

[5] Betts, A., N. Chong, A. F. Donaldson, S. Qadeer and P. Thomson, *GPUVerify: a verifier for GPU kernels*, in: *OOPSLA*, 2012.

[6] Chen, P., Y. Wang, Z. Xin, B. Mao and L. Xie, *Brick: A binary tool for run-time detecting and locating integer-based vulnerability*, in: *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, 2009, pp. 208–215.

[7] Chiang, W., G. Gopalakrishnan, G. Li and Z. Rakamaric, *Formal analysis of GPU programs with atomics via conflict-directed delay-bounding*, in: *NASA Formal Methods, 5th International Symposium, (NFM)*, 2013, pp. 213–228.

[8] Davidson, A., D. Tarjan, M. Garland and J. D. Owens, *Efficient parallel merge sort for fixed and variable length keys*, in: *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–9.

[9] Dietz, W., P. Li, J. Regehr and V. Adve, *Understanding integer overflow in c/c+*, in: *ICSE*, 2012.

[10] Eklund, A., P. Dufort, D. Forsberg and S. M. LaConte, *Medical image processing on the gpu–past, present and future*, Medical image analysis **17** (2013), pp. 1073–1094.

[11] Godefroid, P., M. Y. Levin, D. A. Molnar et al., *Automated whitebox fuzz testing.*, in: *NDSS*, 2008, pp. 151–166.

[12] Granas, A. and J. Dugundji, "Fixed point theory," Springer Science & Business Media, 2003.

[13] Harris, M., S. Sengupta and J. D. Owens, *Parallel prefix sum (scan) with cuda*, GPU gems **3** (2007), pp. 851–876.

[14] Li, G. and G. Gopalakrishnan, *Scalable SMT-based verification of GPU kernel functions*, in: *FSE*, 2010, pp. 187–196.

[15] Li, G., P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh and S. P. Rajan, *GKLEE: concolic verification and test generation for GPUs*, in: *PPOPP*, 2012, pp. 215–224.

[16] Li, P., G. Li and G. Gopalakrishnan, *Parametric flows: automated behavior equivalencing for symbolic analysis of races in cuda programs*, in: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, p. 29.

[17] Liu, W., B. Schmidt, G. Voss and W. Müller-Wittig, *Accelerating molecular dynamics simulations using graphics processing units with cuda*, Computer Physics Communications **179** (2008), pp. 634–641.

[18] Molnar, D., X. C. Li and D. Wagner, *Dynamic test generation to find integer bugs in x86 binary linux programs.*, in: *USENIX Security Symposium*, 2009, pp. 67–82.

[19] Nguyen, H., "Gpu gems 3," 2007.

[20] Nickolls, J., I. Buck, M. Garland and K. Skadron, *Scalable parallel programming with cuda*, Queue **6** (2008), pp. 40–53.

[21] Sengupta, S., M. Harris, M. Garland and J. D. Owens, *Efficient parallel scan algorithms for many-core gpus*, Scientific Computing with Multicore and Accelerators (2011), pp. 413–442.

[22] Tzeng, S. and L.-Y. Wei, *Parallel white noise generation on a gpu via cryptographic hash*, in: *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008, pp. 79–87.

[23] Wang, T., T. Wei, Z. Lin and W. Zou, *Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.*, in: *NDSS*, 2009.

[24] Yablonski, D., *Numerical accuracy differences in CPU and GPGPU codes* (2011), http://www.coe.neu.edu/Research/rcl/theses/yablonski_ms2011.pdf.