Transforming OpenMP to CUDA

Nicolas Merz and AmirHossein Sojoodi

April 2020

1 Introduction

The advancement of both the CPU and GPU, especially within the last 10 years, has largely been in part to improving their parallelization. To increase the ease with which developers can exploit this technology, software libraries must be developed for the hardware; two of the most popular are CUDA for the GPU and OpenMP for the CPU. These two components of a computer's hardware, and subsequently the libraries supporting them, specialize in different tasks and have their own strengths and limitations.

The CPU is generally thought of to be "smarter", meaning it can run more complex instructions, with the downside of having a limited number of cores, whereas the GPU could originally only perform much simpler instructions (such as arithmetic), but it does have many more cores. This makes the CPU excel at doing things that require more control of execution, whereas the GPU is good at things like multiplying large matrices quickly. This means there is not a huge overlap in what they can do, but there are certainly cases where the computation done by a parallelized OpenMP section is simple enough that it can be accomplished in CUDA. This may be desirable if some hardware of a system is being upgraded or changed and a new/more powerful GPU is added. Therefore there is value in a tool that can automatically transform this eligible code from OpenMP to CUDA.

1.1 OpenMP

OpenMP is a shared memory library that focuses on shared-memory multiprocessing on a single multi-core CPU [1]. OpenMP largely uses compiler directives to specify how loops should function in parallel (for instance, the directive "#pragma omp parallel" indicates that the section following it is parallelizable). OpenMP directives create the opportunity to effortlessly exploit multiple cores of the CPU simultaneously. It supports various parallel computing requirements, such as automatic parallelization, efficient reduction, scheduling, granularity management, task splitting and much more. Its simplicity alongside with its rich and powerful features have made OpenMP a popular library over the past years, especially in High-Performance Computing applications.

1.2 CUDA

GPU programs are typically written using the CUDA or OpenCL language [2]. Every CUDA program is divided into Host and Device sections that run on the CPU and GPU, respectively. The NVCC compiler (NVIDIA Compiler Collection) translates the CUDA code into these two major parts. The main part of the code which runs on the GPU is referred to Kernel function. The Kernel function(s) gets called (launched) by the host section and they get executed by many threads in parallel. These threads are the basic components of task granularity in the GPU programming model. They are grouped as thread blocks, and each these blocks are grouped into thread grids. Managing the thread organization is done by the developer and usually is a time consuming task. Typically, to run a task on the GPU, one needs to do the following three steps in consequence:

1. Manage necessary data initialization and copy the data required by the kernel from the host to the GPU device memory.
2. Create/launch the kernel.
3. Return the resulting data from the device back to the host memory, and free the allocated pointers.

It is worth mentioning that modern GPUs' run-time system handles the first and the third item automatically, and only the second step should be done specifically in the source code.

2 Problem Overview

There are multiple research studies and projects on source-to-source compilation from OpenMP or from general C programs to CUDA source code. However, to the best of our knowledge, the TXL language is not used in any similar transformation. Therefore, we implemented a specific transformation tool to translate a simplified C OpenMP source code to its CUDA equivalent using the TXL language. As it was mentioned in the previous section, some modern NVIDIA GPUs (that have compute capability higher than 3.5) provide a feature called UnifiedMemory that enables developers to access the same data from the host and device without worrying about where the data is located at the moment. To narrow the scope of this project, we will focus on the GPUs that have this feature.

Moreover, we assume that the input OpenMP source code has correct syntax and it is parallelizable and valid. This assumption means that the code does not have any semantic problems due to race condition cases and atomic operations.

3 Selected Problem Subset and Implementation

3.1 Basic Case

With such a large variety of parallelization options and edge cases, a fairly small subset was selected given the limited amount of time and manpower. It was decided that the most basic possible case, where a for loop assigns some value for each index of an array, would be selected. In addition, these arrays must use dynamic allocation when they are defined, so that they can easily be converted to use cudaMalloc:

```
.
.
int *array;
array = (int *)malloc(size * sizeof(int));
.
.
#pragma omp parallel
for(int i = 0; i < size; i++)
        array[i] = i;
.
.
free(array);
.
.
```

With this most basic case, there are a few changes that need to be made. The simplest of these are changing the basic malloc and free function calls. Instead of setting a pointer equal to a malloced block of memory (with the size * sizeof(int)) argument), the new cudaMallocManaged function has a void return and now takes a reference to the array pointer:

```
int *array;
cudaMallocManaged(&array, size * sizeof(int));
```

Which is accomplished by extracting the name of the array that is being assigned to, placing it in the cudaMallocManaged function call, and putting a "&" in front of it.

```
replace $ [simple_statement]
        Id[id] Op[assignment_operator] CastOp[cast_operator] Expr[unary_expressi
        deconstruct * Expr
                'malloc ( OrigArgs [list argument_expression] )
        by
                'cudaMallocManaged ( '& Id, OrigArgs)
```

At the bottom end of the code, similar treatment is given to the free function, but is even simpler, and is just statically changed from "free(array)" to "cudaFree(array)".

By far the most complex part of this basic case is implementing the actual CUDA kernel. While in the original OpenMP code a for loop with an "#pragma omp parallel" tag is sufficient, CUDA requires a kernel function. For all cases, it was assumed that there would only be one kernel function to help in simplifying the problem (as there could be cases where there would be multiple parallel OMP for loops that would each need their own kernel function). The for loop must be deconstructed, taking the variable names and statements that are necessary, and reconstructed as a global kernel function and a call to this function, as such:

```
...

#define BLOCK_SIZE 64

__global__ void kernel(int *array, int s){

        int i = blockIdx.x * blockDim.x + threadIdx.x;

        if (i < s) {
                array[i] = i;
        }
}

...

kernel<<<(size - 1)/BLOCK_SIZE + 1, BLOCK_SIZE>>>(array, size);
cudaDeviceSynchronize();

...
```

In the original code, the "size" variable was used as the target for the i index iterator, which enforced the bounds of the array. In CUDA, what is instead used is an if statement based on a calculated index, which is determined using the product of block index and block dimension plus the thread id, similar to how addressing a 2D static array could be done in C (which would use the width of each line instead of block dimensions). If the calculated index is within these bounds, the assignment to the calculated array index is performed.

Some of the transformation for this basic case can be done statically. The BLOCK_SIZE definition, for simplicity's sake, was set to a constant 64. In addition, all of the kernel definitions and function calls (including the cudaDeviceSynchronize() call) were done statically except for where the "size" variable shows up. This size is determined dynamically by examining the conditional_expression portion of the main for loop statement. So as not to have too many cases to consider, this conditional_expression portion MUST take the form of "i < {some_size}", where some_size is a shift_expression and is an integer:

```
deconstruct ForLoop
```

```
'for (D [decl_specifiers] Init[list init_declarator+] '; Condition[conditional_e
        SubStatement[sub_statement]

...

deconstruct Condition
    E1 [shift_expression] Op [relational_operator] E2 [shift_expression]
```

This shift_expression is taken to be the size supplied to the triple angle brackets portion as well as the argument after the array argument in the function call (which in this test case was just the int variable "size"). Finally, the block that appears in the for loop is extracted and placed under the if statement in the kernel function definition.

The TXL grammar for the C language had to have a small modification to support the triple angle brackets, as these are specific to CUDA and do not at all appear in standard C. This was fairly straightforward, as all that had to be done was to redefine the postfix_extension non-terminal of the grammar and add another parseable form. This was copy and pasted from the option for the postfix_extension that appears after the majority of function calls, but with the triple angle brackets added with an argument_expression list in the middle.

```
redefine postfix_extension
        ...
    |   [SPOFF] '<'<'< [SPON] [list argument_expression] [SPOFF] '>'>'> '( [SPON
#ifdef GNU
            [opt dotdot]
#endif
        ')
end redefine
```

With all these transformations programmed, there was a successful conversion between the first sample of C and corresponding sample of CUDA code. The original C code was:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000

int main(){

        int *array;
        int size = SIZE;
        array = (int *)malloc(size * sizeof(int));
```

```
        #pragma omp parallel
        for(int i = 0; i < size; i++){
                array[i] = i;
        }

        // A simple validity test
        printf("Array[%d] = %d", size - 1, array[size - 1]);

        free(array);

        return 0;
}
```

Whereas the CUDA code that was generated after transformation was:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define BLOCK_SIZE 64

__global__ void kernel(int *array, int s){

        int i = blockIdx.x * blockDim.x + threadIdx.x;

        if (i < s) {
                array[i] = i;
        }
}

int main(){

        int *array;
        int size = SIZE;

        cudaMallocManaged(&array, size * sizeof(int));

        // Compute ceil(size/blockSize)
        kernel<<<(size - 1)/BLOCK_SIZE + 1, BLOCK_SIZE>>>(array, size);
        cudaDeviceSynchronize();

        // A simple validity test
        printf("Array[%d] = %d", size - 1, array[size - 1]);

        cudaFree(array);
```

```
        return 0;
}
```

3.2 Multiple Arrays

The next step was to support the usage of multiple arrays within the parallelized for loop and thus the CUDA kernel after transformation. The main component that had to be changed to support this functionality was to extract all the arrays that were being used in the for loop block. This was accomplished by creating a global TXL list of Ids. For each id that was found within the for loop block, it was exported to the Ids list, after being filtered in a function NewVarId. This function ensured that each new Id that was being added to the Ids list was not already in that list, to avoid duplicates in the cases where a variable appeared multiple times in the for loop.

```
rule extract_id
    replace $ [id]
                NewVarId [id]
    import Ids [repeat id]
    export Ids
        Ids [add_new_var NewVarId]
    by
        NewVarId
end rule

function add_new_var NewVarId [id]
        replace [repeat id]
                Ids [repeat id]
        where all
                NewVarId [~= each Ids]
        by
                Ids [. NewVarId]
end function
```

This list of Ids is sufficient for the function call to the kernel function, as calling functions does not require knowing the types of the variables. Now, instead of statically adding "array" to the kernel function call, this dynamically created Ids list is placed instead. Take an instance where there are 3 arrays being used in the for loop. If the original for loop was:

```
#pragma omp parallel
for(int i = 0; i < size; i++){
        array1[i] = i;
    array2[i] = i;
        array3[i] = i;
}
```

The resulting CUDA kernel call should look like:

```
kernel<<<(size - 1)/BLOCK_SIZE + 1, BLOCK_SIZE>>>(array1, array2, array3, size);
```

Where the list of arrays in the call have been extracted from the for block. Finally, the shift_expression that was discussed earlier is appended to this list.

There remains the issue of the function definition, as this requires the supplied arguments to be typed. The typing will be extracted by looking at each variable declaration in the program, and if it matches one of the ids in the Ids list, it is exported into a global Declarations list. For instance, in the above example, array1's declaration would look something like: "int *array1;". The TXL program would find the declaration, import the Ids list, and see if there is any match between the id found in the declaration and an id in the list, exporting it if there is. However, instead of exporting it as a declaration, it is instead exported as an argument_declaration, which is identical except it omits the semicolon.

```
rule extract_array_declarations
    replace $ [declaration]
        DoS [declaration]
    deconstruct * [declaration] DoS
        Spec [decl_specifiers] Decl [declarator] ';
    construct ArgDecl [argument_declaration]
        Spec Decl
    deconstruct * [id] Decl
        Id [id]
    import Ids [repeat id]
    deconstruct * [id] Ids
        Id
    construct ArgExpr [list argument_expression]
        Id
    import ArgExpressions [list argument_expression]
    export ArgExpressions
                _ [construct_arg_expressions_1 ArgExpressions ArgExpr]
                [construct_arg_expressions_2 ArgExpressions ArgExpr]
    import Declarations [list argument_declaration]
    export Declarations
        Declarations [, ArgDecl]
    by
        DoS
end rule

function construct_arg_expressions_2 ArgExpressions[list argument_expression] Ar
        replace [list argument_expression]
                %
        deconstruct ArgExpressions
```

```
                Expressions [list argument_expression+]
        by
                ArgExpressions[, ArgExpr]
end function

function construct_arg_expressions_1 ArgExpressions[list argument_expression] Ar
        replace [list argument_expression]
                %
        deconstruct ArgExpressions
                Empty [empty]
        by
                ArgExpr
end function
```

This made the process fairly simple, as removing the semicolon is trivial and doing so puts the declaration in the exact form that is required by the function definition. After all these declarations are found, the shift_expression and its static int typing are appended to the declaration list, which is then placed in the brackets of the function definition.

The final change was to make sure only arrays that were going to be used inside the kernel function used cudaMallocManaged and cudaFree. This was simply done by including a check in the static replacement, and first ensure that the Id in the malloc/free statement matched one in the Ids list. Like before, a C sample and its corresponding CUDA sample were prepared, which was then used to test for the proper functionality of the TXL program. The original C code was:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define DUMMY 1369

int main(){

        int *array1;
        int *array2;
        float *array3;
        float *array4;
        int size = SIZE;
        array1 = (int *)malloc(size * sizeof(int));
        array2 = (int *)malloc(size * sizeof(int));
        array3 = (float *)malloc(size * sizeof(int));
        array4 = (float *)malloc(size * sizeof(float));
```

```
        #pragma omp parallel
        for(int i = 0; i < size; i++){
                array1[i] = i;
        array2[i] = array1[i] + 1;
        array2[i] = array2[i] + DUMMY;
                array3[i] = array2[i] / (array1[i] + 0.5) ;
        }

        // A simple validity test
        printf("Array3[%d] = %f", size - 1, array3[size - 1]);

        free(array1);
    free(array2);
    free(array3);
        free(array4);

        return 0;
}
```

Whereas the CUDA code was:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define DUMMY 1369
#define BLOCK_SIZE 64

__global__ void kernel(int *array1, int *array2, float *array3, int s) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < s) {
        array1[i] = i;
        array2[i] = array1[i] + 1;
        array2[i] = array2[i] + DUMMY;
        array3[i] = array2[i] / (array1[i] + 0.5);
    }
}


int main(){

        int *array1;
    int *array2;
    float *array3;
    float *array4;
```

```
    int size = SIZE;

    cudaMallocManaged(&array1, size * sizeof (int));
    cudaMallocManaged(&array2, size * sizeof (int));
    cudaMallocManaged(&array3, size * sizeof (int));
    array4 = (float *) malloc (size * sizeof (float));

    kernel<<<((size) - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(array1, array2, array3, si
    cudaDeviceSynchronize();

    printf ("Array3[%d] = %f", size - 1, array3[size - 1]);

        cudaFree(array1);
    cudaFree(array2);
    cudaFree(array3);
    free(array4);

        return 0;
}
```

Do note how there are in fact 4 arrays declared/defined, but only the first three use the cudaMalloc and cudaFree functions, since the fourth did not appear in the for loop.

3.3 Non-Array Variables

The next case should be where there are non-array variables within the for loop. Fortunately, since the previous modification just looks at ids that appear in the for loop, this case is already handled without any further work. To show that this variable extraction is fairly robust, there should be a sample where non-array variables are extracted from the for loop and properly added to the two argument lists. The C sample that was created is:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define PI 3.14

int main(){

        int *array1;
        int *array2;
    double *array3;
        float *array4;
        float *array5;
```

```
        float foo;
        float bar;

        int size = SIZE;
        array1 = (int *)malloc(size * sizeof(int));
        array2 = (int *)malloc(size * sizeof(int));
        array3 = (double *)malloc(size * sizeof(double));
        array4 = (float *)malloc(size * sizeof(float));
        array5 = (float *)malloc(size * sizeof(int));

        foo = 2 * PI;
        bar = 3 * PI;

        #pragma omp parallel
        for(int i = 0; i < size; i++){
                array1[i] = i;
        array2[i] = array3[i] + array1[i];
        array2[i] = array2[i] + 10;
                array4[i] = foo * array2[i];
        }

        array5[0] = array4[0] * bar;
        // A simple validity test
        printf("Array5[0] = %f", array5[0]);

        free(array1);
    free(array2);
    free(array3);
        free(array4);
        free(array5);

        return 0;
}
```

And the corresponding transformed CUDA code looks like:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define PI 3.14
#define BLOCK_SIZE 64

__global__ void kernel(int *array1, int *array2, double *array3, float *array4,
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < s) {
```

```
        array1[i] = i;
        array2[i] = array3[i] + array1[i];
        array2[i] += 10;
                array4[i] = foo * array2[i];
    }
}


int main(){

        int *array1;
    int *array2;
    double *array3;
    float *array4;
    float *array5;
    int size = SIZE;

    cudaMallocManaged(&array1, size * sizeof (int));
    cudaMallocManaged(&array2, size * sizeof (int));
    cudaMallocManaged(&array3, size * sizeof (double));
    cudaMallocManaged(&array4, size * sizeof (float));
    array5 = (float *) malloc (size * sizeof (float));

    kernel<<<((size) - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(array1, array2, array3, ar
    cudaDeviceSynchronize();

    printf ("Array3[%d] = %f", size - 1, array3[size - 1]);

        cudaFree(array1);
    cudaFree(array2);
    cudaFree(array3);
    cudaFree(array4);
    free(array5);

        return 0;
}
```

3.4 Multiple for Loops

The final test case was to ensure the TXL program would ignore normal for loops (those that don't have the "#pragma omp parallel" tag above them), as well as to test if variable declarations inside a for loop block were functional. Once again, this did not require any modifications to the TXL code that had been developed up until this point, since the kernel extracting rule only finds for loops preceded by the tag, and the block is copied verbatim into the generated

kernel function. The new sample C program has a second for loop that does not have an OMP tag, as well as a tmp variable that is declared within the for loop block that should appear in the CUDA kernel function. The C code was the following:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define PI 3.14

int main(){

        int *array1;
        int *array2;
    double *array3;
        float *array4;
        int *array5;
        double foo;
        int size = SIZE;

        array1 = (int *)malloc(size * sizeof(int));
        array2 = (int *)malloc(size * sizeof(int));
        array3 = (double *)malloc(size * sizeof(double));
        array4 = (float *)malloc(size * sizeof(float));
        array5 = (int *)malloc(size * sizeof(int));
        foo = PI * 2;

        #pragma omp parallel
        for(int i = 0; i < size; i++){
                array1[i] = i;
        array2[i] = array3[i] + 1;
                int tmp = array2[i] + 10;
                array4[i] = foo * tmp;
        }

        for(int i = 0; i < size; i++){
                array3[i] = 0.1;
                array4[i] = i;
        }

        // A simple validity test
        printf("Array[%d] = %d", size - 1, array1[size - 1]);

    free(array1);
```

```
    free(array2);
    free(array3);
    free(array4);
    free(array5);

        return 0;
}
```

And the equivalent CUDA code is:

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>

#define SIZE 10000
#define PI 3.14
#define BLOCK_SIZE 64

__global__ void kernel (int *array1, int *array2, double *array3, float *array4,
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < s) {
        array1[i] = i;
        array2[i] = array3[i] + 1;
        int tmp = array2[i] + 10;
        array4[i] = foo * tmp;
    }
}

int main(){

        int *array1;
        int *array2;
    double *array3;
    float *array4;
    int *array5;
    double foo;
    int size = SIZE;

    cudaMallocManaged (&array1, size * sizeof(int));
    cudaMallocManaged (&array2, size * sizeof(int));
    cudaMallocManaged (&array3, size * sizeof(double));
    cudaMallocManaged (&array4, size * sizeof(float));
    array5 = (int *) malloc (size * sizeof(int));
    foo = PI * 2;

    kernel<<<((size) - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(array1, array2, array3, ar
    cudaDeviceSynchronize ();
```

```
        for (int i = 0; i < size; i++) {
        array3[i] = 0.1;
        array4[i] = i;
    }

    printf ("Array[%d] = %d", size - 1, array1[size - 1]);
    cudaFree(array1);
    cudaFree(array2);
    cudaFree(array3);
    cudaFree(array4);
    free(array5);

        return 0;
}
```

4 Lessons Learned

4.1 Scoping

The amount of work and consideration of information extraction and transformation really put into perspective how quickly a TXL project can ramp up in intricacy. A fair number of man hours were dedicated by two people which only supported and was tested on simple OpenMP code, so even after creating what we thought is a pretty robust set of transformation rules, the odds of it working right away on a real software project is slim. This testing on real software would require another large number of man hours to identify problems, fix bugs, and add support to edge cases that were overlooked. One such case that would be relatively common that is not currently handled would be nested parallelized loops, which would make generating proper indexing in the CUDA kernel more difficult.

4.2 C vs OpenMP

With the completion of the project also came an understanding of how much relying on the source code being OpenMP code simplifies the transformation process. The original project plan was to transform ANY C for loops to CUDA. There are certainly a large number of C programs that would do some sort of index-wise assignment in an array that would be simple enough for automatic transformation to CUDA.

But the problem lies in making this distinction of appropriate code to transform; when a programmer is manually making a conversion from some single-threaded code to parallel code, they would have a decent understanding of which parts of the problem are easily parallelizable and which aren't. However, when

automating this process, many more rigorous and complicated checks would need to be made to ensure the code matches a parallelizable form. One of these, for instance, would be if an array assignment depends on another element within the array. This can be done in CUDA, but it is a lot more nuanced than just transforming simple, independent array assignment, and optimizing it often depends on how data is stored.

OpenMP has the valuable benefit that it is programmed with parallelization in mind. While of course there are a lot of technology-specific considerations that need to be made for creating the transformation rules, the fact that OpenMP is the source should alleviate at least some of the theoretical parallelization problems. For example, if the for loop is tagged with a "#pragma omp parallel", the simplicity of chosen keywords should indicate to the TXL programmer that the for loop inside is one of the simplest OpenMP blocks possible. Other keywords such as these should help determine what kind of transformation is necessary so that the equivalent CUDA code deals with any of the same specializations that the omp tags signify.

4.3 Summary

Overall, this was a great project for an introduction to TXL. With guidance, the project was scoped to an appropriate size (the original being much broader, and likely would have not been completed). It created the interesting problems of having to extract variables into a list (watching for duplications), matching keywords to those in a list, prepending blocks of code, among many others. The skills learned for this type of project would translate quite well to any other TXL project, especially to a fact extraction based project.

5 Conclusion

A fairly successful transformation program was developed for the limited subset of the overall OpenMP to CUDA transformation problem. A complete simple case, where a parallelized for loop was converted to a CUDA kernel, was implemented fairly robustly. The transformation supported as many arrays, and furthermore variables, as needed, with any code within the for block being completely copied over (assuming there were not more nested loops inside). In addition, it successfully generated both declarative and normal argument lists dynamically by examining which variables were used within the for loop.

Despite the success, only a very simple and limited case can be transformed. The immediate next steps would be to support multiple kernels (but still for the simple parallel for case). Some type of TXL global variable would need to keep track of how many kernels have been generated (denoted by n, for example), so that the next one could be named something along the lines of "kernel(n+1)".

Once this multi-kernel support has been added, the next step would be to support more OpenMP keywords. An obvious one would be the "reduction" keyword, which signifies that the parallelized loop performs a reduction operation (such as summing all the elements in a list).

The basic parallel case is a fairly simple one, one that could be considered a prototype showing that this type of transformation could even work at all. These extra keywords however will all require quite a bit of work as the parallelization becomes more specialized as they are added. For instance, with the reduction case, there must be care taken in the manual programming of a CUDA kernel for the operation to avoid problems such as warp divergence (where threads in the same "warp", which is a block of threads executing the same code, end up doing different tasks). These all need to be considered when performing the transformation, which may involve a huge set of edge cases that need to be handled, some of which could easily be overlooked. This would greatly increase the complexity and reduce the maintainability of this TXL system.

References

[1] *Openmp application programming interface*, 2018. [Online]. Available: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

[2] *Cuda c programming guide*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract`.