

AutoGPU: Automatic Generation of CUDA Kernel Code

Alexandros-Stavros Iliopoulos
Aristotle University of Thessaloniki
`iliopoulos.alexandros@gmail.com`

June 7, 2011

Σιγά, μωρέ παλιόκοσμε, που θα κάτσω να προγραμματίσω.
—Νίκος Καρούζος [1926–1990]

Contents

1	Representing the kernel	2
1.1	Abstract Syntax Trees	2
1.2	AST implementation in <i>AutoGPU</i>	3
1.2.1	The <i>expr</i> , <i>cond</i> and <i>lhs</i> datatypes	3
1.2.2	The <i>vartype</i> and <i>fltprec</i> datatypes	4
1.2.3	The <i>assignmode</i> datatype	5
1.2.4	The <i>statement</i> datatype	5
2	Optimizing the kernel	7
2.1	The Arguments of the Kernel	9
2.2	Miscellaneous Tools	9
2.3	Transformations on the AST	10
2.3.1	Sparse vs. dense indexing scheme	10
2.3.2	Accumulating partial results	11
2.3.3	Caching through the <code>__shared__</code> memory space	11
2.3.4	Unrolling the loops	12
2.3.5	Staying within array bounds	13
2.4	Translation to CUDA source code	14
2.4.1	Translating the AST	14
2.4.2	Translating syntactic parameters of the AST	15
2.5	Interaction with the Operating System	15
2.6	AST Optimization	16
3	<i>AutoGPU</i> running instructions	18
3.1	Source code files & compilation	18
3.2	Input, parameters and execution	18
3.3	Testing platform development	23
4	Examples	25
4.1	Matrix Multiplication: <code>MATMUL</code>	25
4.1.1	Without conditionals	26
4.1.2	With conditionals	35
4.2	Direct 2D Convolution: <code>CONV</code>	46
4.2.1	5×5 convolution template	47
4.2.2	11×11 convolution template	57
5	Conclusions and future considerations	65
	Appendices	68
A	Examples' testing platforms	68
	Bibliography	82

List of Tables

1.1	<i>statement</i> datatype constructors	6
2.1	<i>ParFor</i> loop counter value assignment	14
3.1	Appropriate CUDA execution configuration for 1D & 2D kernels	23
4.1	The computation units of AUTH' <i>quatro</i> server	25
4.2	Optimal and reference MATMUL kernels specifications	33
4.3	MATMUL kernel (no conditionals) output's maximum absolute error	33
4.4	MATMUL kernel execution times and relative speed-up for 2048×2048 matrices . . .	35
4.5	MATMUL kernel FLOPS and relative speed-up for 2048×2048 matrices	35
4.6	Optimal and reference MATMUL kernels (any matrix size) specifications	44
4.7	MATMUL kernel (with conditionals) output's maximum absolute error	46
4.8	MATMUL kernel execution times and relative speed-up for 3000×3000 matrices . . .	46
4.9	MATMUL kernel FLOPS and relative speed-up for 3000×3000 matrices	46
4.10	Optimal and reference CONV kernels specifications (5×5 template)	57
4.11	CONV kernel output's maximum absolute error (5×5 template)	57
4.12	CONV kernel (5×5 template) execution times and relative speed-up for 8192×8192 matrices	59
4.13	CONV kernel (5×5 template) execution times and relative speed-up for 8192×8192 matrices	59
4.14	Optimal and reference CONV kernels specifications (11×11 template)	59
4.15	CONV kernel output's maximum absolute error (11×11 template)	60
4.16	CONV kernel (11×11 template) execution times and relative speed-up for 8192×8192 matrices	60
4.17	CONV kernel (11×11 template) execution times and relative speed-up for 8192×8192 matrices	60

List of Figures

1.1	A C source code segment and its AST representation, as implemented by the <i>AutoGPU</i> program	3
2.1	Structure relations in the <i>AutoGPU</i> program	8
3.1	Example of specifying a user-defined AST as the <i>AutoGPU</i> input program	19
3.2	Example of specifying a kernel's arguments and constants in the <i>AutoGPU</i> program	20
3.3	Example of specifying a desired loop configuration in the <i>AutoGPU</i> program . . .	20
3.4	Example of specifying the execution configuration for a program with 2 command-line arguments, compiled in the CUDA SDK environment.	21
3.5	Example of how to specify generation of a set of powers of 2 in the <i>AutoGPU</i> program	22
3.6	Example of how to specify generation of a set of divisors of 12 in the <i>AutoGPU</i> program	22
4.1	mm_s: Serial MATMUL kernel	26
4.2	mm_0: Vanilla parallel MATMUL kernel	26
4.3	mm_1: Staged accumulation parallel MATMUL kernel	27
4.4	mm_2: Staged & cached parallel MATMUL kernel	27
4.5	mm_autogpu: Optimal MATMUL kernel	30
4.6	Execution times of the input, intermediate and final MATMUL kernels	31
4.7	Performance of the input, intermediate and final MATMUL kernels	31
4.8	mm_wolfe: Michael Wolfe's kernel	33
4.9	Execution times of the reference and optimal MATMUL kernels	34
4.10	Performance of the reference and optimal MATMUL kernels	34
4.11	mm_cond_0: Vanilla parallel MATMUL kernel (any matrix size)	36
4.12	mm_cond_1: Staged accumulation parallel MATMUL kernel (any matrix size)	37
4.13	mm_cond_2: Staged & cached accumulation parallel MATMUL kernel (any matrix size)	38
4.14	mm_cond_autogpu: Optimal MATMUL kernel (any matrix size)	40
4.15	Execution times of the input, intermediate and final MATMUL kernels with conditionals	42
4.16	Performance of the input, intermediate and final MATMUL kernels with conditionals	42
4.17	mm_cond_wolfe: Michael Wolfe's kernel (any matrix size)	44
4.18	Execution times of the reference and optimal MATMUL kernels with conditionals . .	45
4.19	Performance of the reference and optimal MATMUL kernels with cinditionals	45
4.20	conv_s: Serial CONV kernel	47
4.21	conv5_0: Vanilla parallel CONV kernel (5×5 template)	48
4.22	conv5_1: Staged accumulation parallel CONV kernel (5×5 template)	48
4.23	conv5_autogpu: Optimal CONV kernel (5×5 template)	56
4.24	Execution times of the input, intermediate and final CONV kernels (5×5 template)	58
4.25	Performance of the input, intermediate and final CONV kernels (5×5 template) . .	58
4.26	conv5_flcc: FLCC library kernel (5×5 template)	61
4.27	Execution times of the the reference and optimal CONV kernels (5×5 template) . .	62
4.28	Performance of the reference and optimal CONV kernels (5×5 template)	62
4.29	Execution times of the input, intermediate and final CONV kernels (11×11 template)	63
4.30	Performance of the input, intermediate and final CONV kernels (11×11 template) .	63
4.31	Execution times of the reference and optimal CONV kernels (11×11 template) . . .	64

4.32	Performance of the reference and optimal CONV kernels (11×11 template)	64
A.1	The source code of the testing platform for the MATMUL kernel	71
A.2	The source code of the input generation utility program for the MATMUL kernel . . .	74
A.3	The source code of the testing platform for the CONV kernel	77
A.4	The source code of the input generation utility program for the CONV kernel	81

Abstract

Manual optimization of a CUDA kernel can be an arduous task, even for the simplest of kernels. The CUDA programming model is such that a high performance may only be achieved if memory accesses in the kernel follow certain patterns; further, fine-tuning of the kernel execution and loop configuration may result in a dramatic increase in performance. The number of possible such configurations make it particularly difficult to obtain a truly fast implementation of a given kernel.

To that end, the *AutoGPU* program was developed: A tool that can measure the performance of a kernel for a defined set of configurations in order to decide on the optimal one. Moreover, the *AutoGPU* program can operate on a serial version of a desired kernel, and apply a series of transformations to change the memory access pattern of the kernel, besides deducing the optimal configuration for it.

Hence, a CUDA kernel is automatically generated that may achieve a very high performance, without requiring that the user has knowledge of the CUDA programming model. Of course, tweaking of the generated kernel may still be applicable, but it is expected that it will be minor.

The *AutoGPU* program was developed in the Standard ML language, and this report presents its structure and capabilities.

Chapter 1

Representing the kernel

In order to effectively manipulate the kernel program, an appropriate representation of it must be found that facilitates its analysis and alteration. For this reason, an *Abstract Syntax Tree (AST)* representation of the inspected kernel is adopted. The AST representation is briefly presented in section 1.1.

This AST representation is exposed to the *AutoGPU* program through several Standard ML *datatypes*. These datatypes are shown in section 1.2.

1.1 Abstract Syntax Trees

It is a common technique for compilers to maintain different representations of the source program through different compilation phases. Such a representation is that of the *Syntax Tree*, which is an intermediate representation of the source code one, derived after completion of the parsing phase, in order to separate issues of parsing from issues of semantics. Syntax trees come in two varieties:

Concrete Syntax Trees consist of exactly one leaf per input token and one internal node per grammar rule reduced during the parsing phase. Syntax trees of this kind contain a significant amount of redundant information, as they are largely dependent on the grammar of the source language. As such, they are not ideally suited for semantic analysis.

Abstract Syntax Trees, on the other hand, constitute a representation free of syntactic details: it is an intermediate step after all parsing issues have been dealt with, but before any semantic interpretation takes place.

Further, the AST representation is generally independent of the programming language in which the source code is written (at least for languages pertaining to the same programming paradigm). Since the *AutoGPU* implementation of the AST representation has been developed to map directly to the CUDA language syntax, the *AutoGPU* program can easily be extended to handle any *imperative* source code as input with the use of an appropriate parser.

A graphical example of an AST representation of a code fragment is shown in figure 1.1. The presented AST follows the *AutoGPU* implementation, which is described in the next section.

```

float threshold;
threshold = 0.0;

for (int i = 0; i <= m; ++i) {
  for (int j = 0; j <= n; ++j) {
    if (A[i + j*m] < threshold) {
      A[i + j*m] = threshold;
    }
  }
}

```

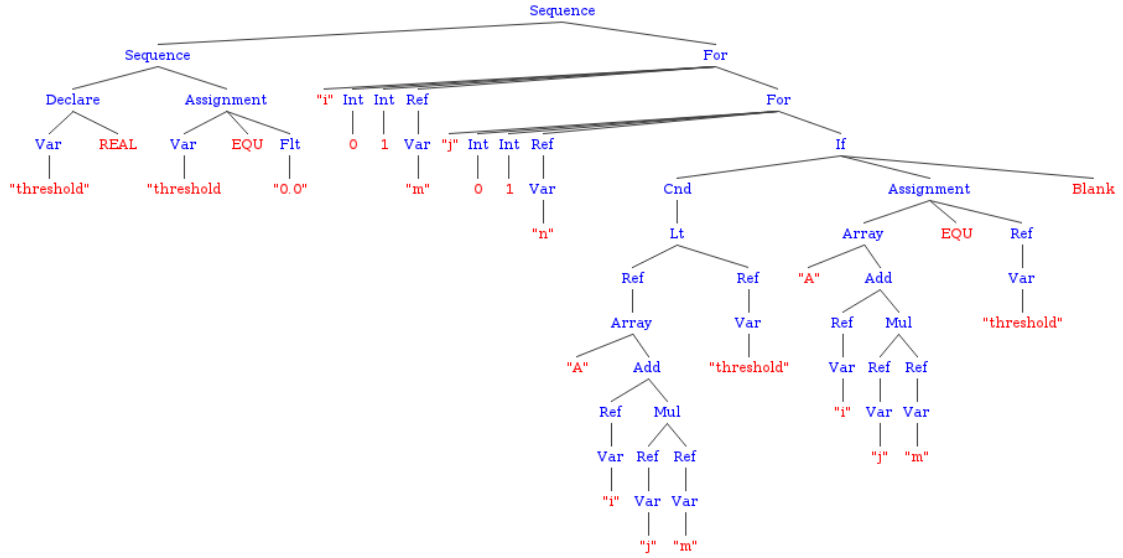


Figure 1.1: A C source code segment and its AST representation, as implemented by the *AutoGPU* program

1.2 AST implementation in *AutoGPU*

An imperative program is made up of several constructs, such as statements which alter the machine state, expressions which are eventually reduced to a value, variable types and qualifiers, etc. For this information to be made available to the *AutoGPU* program, a datatype has to be coded for every corresponding construct. All supported datatypes are discussed in this section.

It should be mentioned that due to the current structure of the *AutoGPU* program (which requires that the *constructors* of each datatype be exposed to most of its modules) the defined datatypes are loaded in the Standard ML top-level environment.

1.2.1 The *expr*, *cond* and *lhs* datatypes

Any expression that can be evaluated (that is, reduced to a value) is represented by the *expr* datatype. Naturally, an expression is defined recursively, and it may be any of the following:

- An *integer number*.
- A *floating-point number* (internally represented as a **string**, in order for arguments of type *expr* in Standard ML functions to maintain the equality property).
- The *width* of the CUDA block, as specified in the kernel's execution configuration.
- An *arithmetic operation* on one or two expressions. The supported operations and corresponding C (and by extension, CUDA) operators are binary addition (+), subtraction (-),

multiplication (*), division (/), modulo (%), left shift (<<), right shift (>>) and unary negation (-).

- A *conditional expression*. This is represented by the *cond* datatype and can be:
 - A *conditional operation* ($=, \neq, >, \geq, <, \leq$) on two expressions.
 - A *logical operation* (NOT, AND, OR) on one or two expressions.
- A *reference*, represented by the *lhs* datatype. The datatype contains at least a string to represent the name of the reference, which can be:
 - A reference to a *variable*, addressed as `var`.
 - A reference to an *array element*, addressed as `array[i]` (where *i* is an expression).
 - A reference to a *pitched 2D-array element*, addressed by the macro `array_at(i,j)` (where *i* and *j* are expressions), which is automatically generated to correspond to the following expression, as the CUDA programming model dictates (arrays are presupposed to be stored in column-major fashion):

```
(*( (arraytype*)( (char*)array + j*pitch_array ) + i ))
```

Because of the C syntax that is used by the CUDA language, a conditional expression is nothing more than one that evaluates to either 0 or 1; this is the only essential difference of the *cond* datatype operations and the *expr* ones. On the other hand, the *lhs* datatype is distinct in that an expression cannot be used in the left-hand-side of an assignment operation (this also means that assignments to memory addresses referenced by pointer expressions are not supported by the *AutoGPU* program).

1.2.2 The *vartype* and *fltprec* datatypes

The type of a variable is necessary for any declaration statement, and representing it is done through the *vartype* datatype. Also, the *vartype* datatype helps provide a way to inspect the type of a variable that appears inside the body of the kernel, for instance one of its formal arguments. The following types are supported:

- The `int` type. This expresses integer numbers.
- The *real* type. This expresses floating-point numbers in general, not taking precision into account.
- The `size_t` type. This is the same as `unsigned int`, but is best used to express memory sizes, as is the case with the CUDA 2D array pitches.
- The *pointer to vartype* type. This type's definition is recursive, therefore pointers to pointers (and so forth) can be defined.

Real (i.e. floating-point) number types differ in their precision, and are supported through the *fltprec* datatype. It is presupposed that all real variables in a kernel will be of the same precision, hence this datatype works like a “global switch” for all *real* variables in the kernel. The following floating-point number types are supported:

- The `float` type.
- The `double` type.
- The `long double` type.

1.2.3 The *assignmode* datatype

As the C or CUDA syntax allows, the assignment operator = can be augmented with any other binary operator, meaning that the value that is assigned to the left-hand-side reference is the result of applying the operator to it and the value the right-hand-side evaluates to. Specifying such an operator is done through the *assignmode* datatype, which is used solely for that purpose. This datatype can take any of the following values:

- EQU. Signifies absence of a binary operator.
- ADD. The addition operator +.
- SUB. The subtraction operator -.
- MUL. The multiplication operator *.
- DIV. The division operator /.
- MOD. The modulo operator %.
- SHL. The “left shift” operator <<.
- SHR. The “right shift” operator >>.

1.2.4 The *statement* datatype

A program represented by an AST is essentially a single recursive statement. Each statement is constructed by information which varies according to the corresponding source statement. This information is referred to as the statement’s *children*, in accordance with the AST structure. The defined statements are outlined in table 1.1.

Apparently, the input AST is expected to be constructed by a subset of these statements, it being a representation of a serial kernel. The *DeclareShr* and *Sync* statements are automatically generated during the AST optimization process; and a *ParFor* statement in serial code may simply be a for loop preceeded by an appropriate directive (e.g. `#pragma simd` or `//SIMD`).

ParFor	A <i>parallel for</i> loop. Assigns an index to each thread, according to its position within the grid and block, as well as some other parameters (i.e. the loop's <i>unroll factor</i>), and “kills” redundant threads.	
	<i>Children:</i>	
	string	The name of the index variable.
	expr	The index of the first element of the loop.
	expr	The stride of the loop.
	expr	The index of the last element of the loop.
	bool	True/false indicates a sparse/dense indexing scheme for this loop (see subsection 2.3.1).
	statement	The body of the loop.
For	A <i>for</i> loop.	
	<i>Children:</i>	
	string	The name of the index variable.
	expr	The index of the first element of the loop.
	expr	The stride of the loop.
	expr	The index of the last element of the loop.
	statement	The body of the loop.
If	An <i>if-then-else</i> conditional statement. Existence of an <i>else</i> part is mandatory.	
	<i>Children:</i>	
	expr	The predicate expression.
	statement	The body of the <i>then</i> part.
	statement	The body of the <i>else</i> part.
While & DoWhile	A <i>while</i> / <i>do-while</i> loop.	
	<i>Children:</i>	
	expr	The predicate expression.
	statement	The body of the loop.
Sequence	Two successive statements.	
	<i>Children:</i>	
	statement	The first statement.
	statement	The second statement.
Assignment	An assignment of the form $lhs [op] = expr$, where <i>op</i> is an optional binary operator (see subsection 1.2.3).	
	<i>Children:</i>	
	lhs	The left-hand-side of the assignment.
	assignmode	An optional binary operator which specifies the way that the left-hand-side reference's initial value affects the assigned value.
	expr	The right-hand-side of the assignment.
Declare & DeclareShr	An automatic / <code>__shared__</code> variable or array declaration.	
	<i>Children:</i>	
	lhs	The declared variable or array. In the case of array declaration, the index field of the <i>Array</i> is interpreted as the size of the array.
	vartype	The type of the declared variable or array.
Sync	The <code>__syncthreads()</code> CUDA intrinsic function.	
	<i>No children.</i>	
Blank	An empty statement. Can be used to express absence of the <i>else</i> part in an <i>if-then-else</i> conditional statement.	
	<i>No children.</i>	

Table 1.1: *statement* datatype constructors

Chapter 2

Optimizing the kernel

The process of performing semantic analysis and applying transformations to the AST representation of the kernel in order to optimize it can be roughly divided in 6 areas, which are not independent of one another:

- i. Identifying certain syntactic parameters of the kernel, such as its name, its formal arguments, the constant values it manipulates, and which arrays should be accessed specially.
- ii. Having a set of tools to traverse the AST, mine data from it, access the expressions therein, etc., as well as being able to access the kernel's parameters (i.e. domain dimension, kernel block width, and thread unrolling factors).
- iii. Performing transformations on the AST, in order to yield a faster (and equivalent) version of it.
- iv. Obtaining the kernel's CUDA source code from its AST representation and its parameters.
- v. Interacting with the underlying OS to create the source code file, compile it and run it.
- vi. Evaluating the performance of each successive version of the AST and conducting the optimization process.

The functions that implement all these procedures are grouped in structures which correspond to each of the aforementioned areas. The structure relations schema is shown in figure 2.1; an arrow means that the structure at its end is the output of a Standard ML functor that receives the structure at its start as a parameter. Also, as was explained before, the Standard ML top-level environment is enhanced with the datatypes presented in Chapter 1. This is done so as to expose the datatype constructors' representation to the structures, to ease the AST manipulation process.

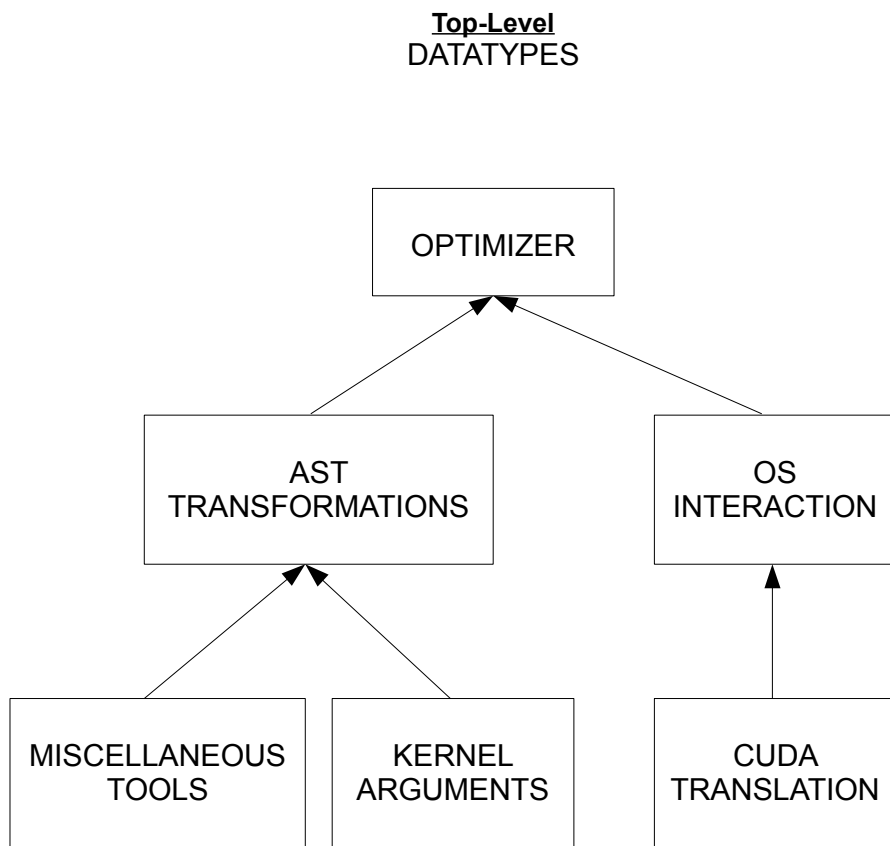


Figure 2.1: Structure relations in the *AutoGPU* program

2.1 The Arguments of the Kernel

Apart from the AST of the serial kernel, we expect, naturally, to be supplied also with the arguments that are passed to it, along with their types. These are kept in a mutable list as pairs of (*argname*, *argtype*). Another two such lists exist; one which contains the variables that are stored in `__constant__` memory space, and one which contains the 2D-pitched arrays, for which the access macros must be defined; both of these lists have their members assigned by the *AutoGPU* program after inspecting the AST. Further, a mutable list is provided where the user may specify the bounds of arrays referenced to in the AST, as a pair of (*array name*, *array bound variables list*). This list is used to generate conditional statements to prevent “out-of-bounds” references (see subsection 2.3.5).

Having an argument reside in `__constant__` memory space instead of passing it directly to the kernel can be advantageous in most cases: If all active threads attempt to access the same address in `__constant__` memory space, this is as fast as accessing a register, therefore there is no loss with respect to performance; on the other hand, having less arguments in a kernel call means that fewer registers are allocated for each block, which means that a microprocessor might execute more blocks concurrently than it otherwise would, or that more automatic variables will reside in registers rather than in local memory, which exhibits a high access latency. Moreover, the kernel may fail to launch if a microprocessor does not have enough registers to process at least one block; hence, moving arguments to the `__constant__` memory spaces allows for more unrolling of the parallel loops (since unrolling a SIMD loop may result in more automatic variables being handled by each thread).

A set of functions allow checking whether a given variable (represented by a `string`) belongs to the arguments of the kernel (either formal ones or ones residing in `__constant__` memory space), getting the type of a given variable, or getting the bounds of a given array (if any are specified).

A function is also provided that traverses the AST in order to mark the 2D-pitched array variables that need macro definitions by adding them to the appropriate aforementioned list. Also, for every 2D-pitched array *X* that is marked, a `__constant__` argument `pitch_X` is generated in the relevant list.

Lastly, a reference to the kernel’s name is also kept, in order to produce relevant source code file names, and to properly declare the kernel function.

2.2 Miscellaneous Tools

Certain parameters of the AST and functions on it are independent of a specific optimizing transformation, and may be found useful even outside the scope of any such transformation. These tools will be considered a given throughout the rest of this report.

There are four important parameters of the AST which are stored in three *mutable* values:

- The problem’s *dimension domain*. This is equivalent to the dimension domain of the arrays which are processed according to the AST, and also comprises the upper bound for the number of *ParFors* that can be present in the parallel version of the AST. For example, this is 2 in the case of 2D-matrix multiplication. Currently, 3-dimensional domains are not supported.
- The *width* of the CUDA block, i.e. the number of threads in each kernel execution block. Only 1-dimensional blocks are defined.
- A 2-tuple containing the unroll factors of the *ParFor* loops, in such order so that the first component corresponds to the outermost *ParFor* loop and so on.
- A list containing the variable counters (represented as `strings`) of every *for* loop of the kernel, coupled with an unroll factor for the corresponding loop. Similarly to marking 2D-arrays, this list is automatically derived by a function that inspects the AST.

Apart from the above parameters, a set of simple functions is also provided. These allow the following:

- Checking whether a given variable (expressed as a **string**) is referenced in an expression.
- Replacing all references to an ‘old’ variable in an expression with references to a ‘new’ variable (both expressed as *references*).
- Attempting to evaluate the numerical (integer) value of an expression, and returning it if the expression contains no *references* nor real numbers.
- Checking an expression for occurrences of a set of variables (expressed as a list of **strings**) and returning all such occurrences as a list of *references*.
- Returning all references that occur within an expression.
- Checking whether two lists have any common elements.
- Applying a **string** suffix to a variable *reference*.
- Replacing all occurrences of the *width* expression in a *statement* (i.e. subAST) with its numeric value.
- Traversing the AST of the kernel in search of *for* loops and marking their counter variables in the relevant list, defaulting the unroll factor with a value of 1.
- Returning all possible permutations on the elements of a list as a list of lists.
- Constructing a series of elements, described by the initial element, the total number of elements, and a function to be recursively applied to an element of the series to calculate the succeeding element.
- Constructing all possible double combinations of elements of a set described in the same way as the aforementioned lists.

2.3 Transformations on the AST

The set of transformations that can be applied to any input AST constitute the core of the *AutoGPU* program, as it is through those transformations that an equivalent but faster version of the AST can be obtained. Some transformations aim at producing an equivalent AST that makes better use of the CUDA programming model (see subsections 2.3.1 to 2.3.3), whereas others alter it as indicated by certain parameters (see subsections 2.3.4 to 2.3.5).

Should a transformation of the former kind be of no use with regard to a specific AST, then applying it to this AST will have no effect; therefore these can be safely applied to any input AST.

All supported transformations are discussed in detail in the following subsections.

2.3.1 Sparse vs. dense indexing scheme

A *ParFor* loop is by default interpreted as a *sparsely indexed* one. This means that if the loop’s index is configured to span values *first* through *last* with a stride of *stride*, then only the threads whose index *tid* satisfies the equation $[(tid - first) \bmod stride] = 0$ will execute.

A *densely indexed ParFor*-loop, on the other hand, means that only the first $\left\lfloor \frac{last - first + 1}{stride} \right\rfloor$ threads will execute. Of course, for this change to occur, their index must be redefined according to the formula:

$$tid' = tid \cdot stride + first$$

This transformation is applied if the stride of the *ParFor* loop is greater than a threshold value, dependent on the resident GPU of the system. The threshold value is currently defined empirically as twice the *warp size*, in order to achieve a speedup in the execution of the kernel.

2.3.2 Accumulating partial results

Accesses in the `__global__` memory space exhibit a very high latency, therefore it is desirable to minimize such accesses as much as possible. A common case where this can be done is the case where a variable or array element accumulates a partial result in the body of a *for* loop: Instead of accumulating the result straight into the variable or array, which resides in `__global__` memory, a “staging” variable can be used. In this way, only 2 accesses in the slow `__global__` memory occur, one while initializing the accumulation variable before entering the relevant *for* loop and 1 while assigning the full result in the proper array element after it; otherwise there would be one such access for each *for*-loop cycle.

This “staging” is performed in the fashion explained below:

The AST is traversed from its root. After the last *ParFor* statement is encountered, the staging function searches for *for* loops. If one is encountered, its counter variable is kept track of (by being pushed into a list), and the search now occurs within its body, only with different objectives. Should a nested *for* loop be encountered, it is treated in the same way as the previous one; hence, the counter variable of the outermost *for* loop can be found in the *tail* of the aforementioned list.

Inside the body of a *for* loop, the search is for *Assignment* statements whose left-hand-sides are arrays (or variables residing in `__global__` memory). If such an assignment is found, the *type* of the array or variable is found through the kernel arguments list, and a 4-tuple is obtained, containing:

- the left-hand-side of the argument,
- the type of the array or variable referenced in it,
- a serial number that records how many such assignments have been found, and
- the counter variable of the outermost *for* loop which the *Assignment* statement is dependent on.

This is then pushed into a list which serves to keep track of all staged assignments. The last element of this 4-tuple can be found if the list with the loop counter variables is reversed and then searched for the first one which appears in the right-hand-side of the assignment. Also, the left-hand-side of the assignment becomes a reference to an automatic variable `accX`, where *X* is the aforementioned serial number.

Every 4-tuple of the above nature contains all the necessary information to completely carry out the staging of the corresponding partial accumulation. Thus, after the body of a *for* loop is searched (and perhaps altered), ascertaining if this is the outermost loop which “influences” a now staged assignment becomes possible. In this case, the corresponding accumulation are initialized just before this *for* loop, so that they hold the initial values of their respective variables or array elements; their (accumulated) values are assigned back to their respective references on termination of the *for* loop.

Finally, the accumulation variables are declared in the beginning of the kernel. For that reason, the whole 4-tuple list is kept until the function is fully applied. After the part which corresponds to the body of the innermost *ParFor* statement returns, the appropriate declarations are generated in its beginning.

2.3.3 Caching through the `__shared__` memory space

The CUDA programming model is such that fetching from the `__global__` memory space is optimized when all threads access consecutive elements. Hence, non-consecutive element access should be avoided, if possible. One very common such case is when all threads in a block access the same element; this is especially problematic when there are successive accesses to the same element, e.g. when such an access occurs within the body of a *for* loop.

This problem can be overcome by fetching the element that all threads (in a block) access simultaneously into `__shared__` memory. Moreover, in the case of consecutive unoptimized accesses, as mentioned above, this can be done best by having each thread in the warp fetch one of the elements in question, so that one memory access is performed instead of *warp size* ones. After fetching the elements into `__shared__` memory, simultaneous access to one of these elements by all threads is as fast as accessing a register, thanks to the CUDA “broadcasting” model.

This transformation on the AST is implemented as follows:

The AST is traversed from the root until a *ParFor* statement is found, and its index is stored as the *independent* one. The traversal goes deeper until a second *ParFor* statement is found, whose index is stored as the *dependent* one.

From now on, the traversal continues in search of a *For* statement. Should one be found, its index variable (*idx*) and stride (*str*) are stored, and its “body” (subAST) is checked for *Assignment* statements with a right-hand-side in which an array access occurs, described by an index (or indices) independent of the *independent* index, but dependent of the *dependent* one. If no such access is found, the body of the *For* statement remains unaltered, otherwise the arrays which fulfil the criteria are cached via `__shared__` memory:

The matching assignment statements are replaced by a *For* statement, with *idx* as the index variable, spanning values *idx_s* to *idx_s* + *Width* with a stride of 1; the body of the new *For* statement is the matching assignment, in which any references to the “cachable” arrays are replaced by references to 1D arrays of the same name suffixed with “_c” and indexed by *idx* – *idx_s*.

The outer *For* statement (i.e. the one in whose body the matching assignment statements were found) is also replaced by a new *For* statement, with *idx_s* as the index variable, spanning the same value range as before, but with stride *str* · *Width*, and the following statements serially comprising its body:

- A *Sync* statement.
- Assignment of values to the arrays residing in `__shared__` memory space. These assignments are of the form `array_c[tx] = array_newref`, where `array` is the name of the array that is being cached, `tx` is the thread’s serial number within the block (not the global one), and `array_newref` is the reference to this array that was found to match earlier on, in which references to the *idx* variable are replaced by references to *idx_s* + *tx*.
- A *Sync* statement.
- The rest of the statements in the body of the *For* statement (including the replacements of the matching assignments).

The assignment/initialization of the caching arrays in `__shared__` memory must be enclosed within *Sync* statements to ensure that all threads have access to the properly updated value of these arrays, according to the CUDA programming model.

Lastly, some declarations are generated in the top of the body of the innermost *ParFor* statement: These are the declaration of `tx` as the block serial number of the thread (`int tx = threadIdx.x`), and the declarations of all the `__shared__` arrays with the appropriate type, which is found by matching the array name against the kernel arguments. An example of such a declaration is `__shared__ float Arr_c[WIDTH]`. It should be noted that the declaration of the thread’s index within the block does not occur in the case that no `__global__` references were cached as described in this subsection.

2.3.4 Unrolling the loops

Loop unrolling is a well-known technique used in the pursuit of higher performance. Its goal is to decrease the loop’s overhead by hiding memory fetching latencies or reducing the number of instructions that control the loop, in order to obtain a faster version of it.

Of course, it is possible that unrolling a loop will have a negative rather than positive effect on it, for instance because of increased register usage in a single iteration (or, in the case of *parallel for* loops, increased register allocation for each thread). Hence, it is desirable to determine the best possible unroll factor for each loop.

ParFor loop unrolling

With regard to the CUDA programming model, there is the additional overhead of loading the kernel blocks into the multiprocessors, which is reduced by unrolling the *ParFor* loops, since in that way a block will compute values that “belong” to other blocks, hence fewer blocks are required to execute. This means that the unrolling “policy” will differ between different-level *ParFor* loops:

A loop counter dependent on the thread index within the block will have to be incremented by a value of *width* to point to the next element for this thread, whereas a loop counter dependent only on the block index within the CUDA grid will have to be incremented by 1.

The factor by which each *ParFor* loop is unrolled has to be set in the relevant parameter before applying the unrolling function on the AST, which will unroll all *ParFor* loops (except the ones set to be unrolled by a factor of 1). This function works as follows:

Firstly, the variables that will have to be reproduced because of the unrolling (as many times as the unrolling factor indicates) have to be found. This is done by checking for assignments whose right-hand-sides are dependent on the unrolled variable (loop counter); the variable on the left-hand-side of the assignment will have to be reproduced. This is done recursively, since the reproduction of a variable may impose that another one need to be reproduced as well, and so forth until no more variable names are appended in the list of “to-be-reproduced” variables. Afterwards, using this list of names (i.e. **strings**), the AST is traversed again and a list of *references* is returned; this lists every reference to the variables in question within the AST.

The *ParFor* loop is now ready to be unrolled. Statements in its body are checked to determine if they contain references to the loop counter or to any affected variable, in which case they will have to be reproduced. Any such statement is replaced by a sequence of statements which are the same as it, with the exception that instances of the loop counter or any affected variable will be altered: The loop variable must be incremented *n* times the offset that corresponds to the loop’s level, where *n* is the order of the copy of the statement, and affected variables must be properly reproduced.

The reproduced variables are suffixed with an integer indicating the order of the variable’s copy. Should a variable be reproduced as a result of the unrolling of more than one *ParFor* loop, then it will have the appropriate number of suffixes, with the leftmost one being a result of the outermost affecting loop.

For loop unrolling

The way that “regular” *for* loops are unrolled is essentially the same as with *ParFor* loops. The only difference is that within the body of the unrolled *for* loop, any instance of the loop counter reference in a reproduced statement has to be incremented by the value of the counter’s stride times the statement’s copy order.

The unroll factor for each loop can be found by searching the relevant mutable list, using the name of the loop counter as the search key.

2.3.5 Staying within array bounds

This transformation is different from the ones discussed throughout this section, as it is not an optimizing transformation on the AST. Quite the contrary, it affects the kernel’s performance negatively, since conditional statements may cause threads in a warp to *diverge*: The threads that will not execute a code segment because of the conditional predicate have to be idle while the rest execute it. It is, however, included in the *AutoGPU* program as a facility to the user, as a means of automatically producing a kernel that works correctly regardless of the array sizes being multiples of the CUDA block sizes.

Should the user specify the bounds of any array that is referenced within the AST of the kernel, application of this transformation will result in placing any assignment which contains references to the array within a conditional statement (one for each array dimension), which requires that the relevant index is within the specified bounds. The generation of one conditional statement per array dimension rather than just one which combines all dimensions was chosen so that the unrolling of a variable which is referenced only in one dimension of the array will only reproduce the conditional expression which refers to it.

The user can specify whether these conditionals before array references should be generated during the optimization process through a flag in the *AutoGPU* program (see section 2.6). This is because the optimal parameters of the AST may change after augmenting it with conditional statements.

2.4 Translation to CUDA source code

In order for the performance of a version of the program, expressed in AST representation, to be measured, the program must be translated to source code, compiled, and run. The CUDA source code that must be yielded contains the AST information, as well as some other parameters (kernel and formal arguments declaration, macro definitions, `__constant__` variable declarations).

Variable types are expressed via the *vartype* datatype, hence whenever the type of a variable is mentioned as part of a **string**, the application of a function that returns the **string** representation of a given *vartype* is implied and will not be further commented upon.

2.4.1 Translating the AST

Translation of the AST to CUDA source code is carried out by function `translateKernel`, which returns the resulting source code as a **string**. The translation is done step-by-step following the traversal of the AST from the root, and it is rather straightforward, therefore only a couple of details will be discussed, instead of the way each *statement*.

Expressions are processed recursively and a set of parentheses is generated around any expression before the translation process proceeds to its sub-expressions. This ensures that every expression will be evaluated in a valid order; and any minor optimization concerning values known at compile time or changing the succession of operations in an expression can be expected to be taken care of by the `nvcc` compiler. As far as references are concerned, these are addressed as shown in subsection 1.2.1.

A parameter (`prec`) of the structure which contains the translation function indicates the precision to be used for floating-point numbers. This affects the string representation of the *type* of real variables, and the suffix of the floating-point literals (nothing for *double*, `f` for *float* and `L` for *long double*). Also, regarding numeric literals, care must be taken to express negative numbers with the `-` (minus sign) instead of the `~` (tilde) notation used in Standard ML, as well as ensuring that the `e` in exponential real number notation is lower case.

The only other point of interest is the translation of the *ParFor* statements. First of all, it should be noted that the translation function receives as input not only the AST, but also the dimension of the problem domain. This is maintained throughout the translation process, along with a value indicating the “current” dimension (which is initialized to have the same value as the domain dimension). Every time a *ParFor* statement is processed, the “current” dimension indicator is decremented by one. This combination of “total” and “current” dimensions/levels provides the means to decide the value that should be assigned to the loop counter variable, as can be seen in table 2.1.

$\left(\begin{array}{cc} \text{current} & \text{total} \\ \text{dimension} & \text{dimensions} \end{array} \right)$	Value assigned to loop counter
(2,2)	<code>blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x</code>
(1,2)	<code>blockIdx.y*UNROLL2</code>
(1,1)	<code>blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x</code>

Table 2.1: *ParFor* loop counter value assignment

Of course, after assigning index values for each thread, the redundant ones (as described by the *first*, *stride* and *last* parameters of the *ParFor* loop) need to be deactivated. The threads that are so “killed” are the ones for which the set index *i* satisfies any of the following:

- For dense indexing:
 - $i < first$
 - $i > last$
 - $(i - first) \bmod stride \neq 0$

- For sparse indexing:

$$\cdot i > \frac{\text{last-first}}{\text{stride}} + 1$$

2.4.2 Translating syntactic parameters of the AST

The output produced by translating the AST cannot be of any use without the proper “environment”. This includes the declaration of the kernel’s formal arguments, as well as defining or declaring any needed macros or `__constant__` variables, respectively.

The formal arguments of the kernel are converted to a **string** of all the arguments (*type* and *name*), separated with commas. Placed between parentheses, this constitutes the formal argument list of the kernel. (However, the kernel declaration is still incomplete, since the kernel function’s name and return type must be specified; this is done during the creation of the `.cu` file, described in section 2.5.)

The macro definitions and `__constant__` declarations are straightforward: The necessary macros are `#defined` as outlined in subsection 1.2.1, separated by a newline character, and the constants are declared as `__device__ __constant__ type constant_name;`, separated by a newline character.

2.5 Interaction with the Operating System

Interaction between the *AutoGPU* program and the OS of the system where it executes involves creating the `.cu` source file which contains the CUDA kernel (along with necessary definitions, declarations, etc.), compiling the program’s source code (in which the kernel code is `#included`), and executing the compiled program to measure its execution time. Also a couple of auxiliary functions are provided in this level, which may be used to inject code segments in the program’s source code that are essential for the optimization process to work correctly.

In order to create the `.cu` file containing the kernel code and all other necessary parameters, a plethora of information is required. This information occurs in the kernel `.cu` file in the following fashion:

- The program’s name is used to create an output stream to the file `kernel-program_name.cu`.
- The width of the CUDA blocks is defined as a macro `WIDTH`.
- The ParFor unroll factors are defined as two macros, `UNROLL1` and `UNROLL2`.
- The 2D-pitched array access macros are defined (see subsection 1.2.1).
- The `__constant__` variables are declared, as mentioned in subsection 2.4.2.
- The kernel function is declared as `__global__ void program_name`, followed by the formal argument list, as described in subsection 2.4.2.
- The CUDA code of the kernel is obtained from the AST, as described in subsection 2.4.1, and comprises the body of the kernel function.

It is assumed that the user, apart from the main program or “testing platform” which invokes the kernel has prepared a Makefile for compiling it, thus the compiling process is nothing more than the issuing of a `make -B` command to the OS. (The `-B` option is necessary, because no changes are detected in the target `.cu` file after simply altering the kernel source code, as the latter is `#included` in the former.) Also, the location of the compiled program (relative or absolute) must be specified, as well as any execution arguments to be passed to the program by command line. This information is expressed by two variables in the OS interaction structure (variables `execLoc` and `execArgs`, respectively).

Compilation automatically occurs whenever the program is called to run: After the execution command has been “assembled” using the aforementioned information, the compilation command is issued before executing the program. Finally, the execution time of the kernel is returned.

From the scope of the *AutoGPU* program, it is impossible to measure the execution time of any *part* of the invoked program. Due to this reason, the execution time of the kernel must be measured by the “testing platform” and saved somewhere where it can be inspected externally. The *AutoGPU* program expects the elapsed time to be output as a single line `string` representation of a floating-point number in the “`comptime.agpu`” file in the current folder (i.e. the active one during the optimization process).

The program’s return value is used to indicate whether the kernel executed correctly or its execution failed. This might happen if the multiprocessors are unable to handle the kernel in a certain execution configuration, for instance because the number of threads per block exceeds the target GPU’s limit, or due to lack of the necessary number of registers as a result of unrolling a *parallel for* loop too much. In this case, a *time* representation of `MaxInt` seconds is returned, effectively discarding the faulty kernel configuration from the set of possibly optimal ones.

Programming style may differ greatly between people, and the rather standard setting that the *AutoGPU* program relies upon with regard to measuring the kernel’s execution time could be the cause of unnecessary frustration during coding of the “testing platform” program. The two auxiliary functions mentioned in the beginning of the section aim to overcome this.

Both work in a similar fashion, and the user may trigger their application through the corresponding flags (see section 2.6).

One is provided to generate and inject code to carry out the process of measuring the execution time and saving it as required. The code is injected just before (starting the timer) and after (stopping the timer, and converting and saving its measurement) the invocation of the kernel. The kernel invocation is recognized in the source code as consecutive lines that start with the strings *kernel-name* and `<<<` and end with a semicolon `;`. Also, the libraries `fstream` and `sstream` are loaded, since they are used to convert the time to a `string` and save it in a file.

The other one may be used to generate and inject code to check whether the kernel launched correctly or failed and to return a non-zero value on failure. The kernel invocation is recognized in the way described above and the error checking code is injected just after it. Also, the source code is scanned for a `return 0;` statement after that, which becomes replaced by a conditional that controls returning either 0 (success) or 1 (failure), depending on the error checking outcome.

For an example of a “testing platform” program and the discussed necessary code segments, see Appendix A.

2.6 AST Optimization

All necessary tools for altering and optimizing an AST have been described in the previous sections of this chapter. These need to be combined in a single automatic procedure that will apply the optimizing transformations on the AST with a set of parameter combinations, comparing the results each version yields, in order to obtain the best one.

This is the highest-level operation provided by the *AutoGPU* program, and as such it has access to each of the defined mutable parameters of the AST, and may inspect them at any time to get their current instances. Furthermore, the AST is input in a mutable variable field of the Optimization structure, which is updated as transformations are applied to it. Whenever a kernel file is generated by the *AutoGPU* program it is implied that it is done so using the *current* set of parameters.

It has been mentioned in the previous section that it is possible to specify whether the *AutoGPU* program should insert code segments that perform the timing and kernel fail checking processes in the main program. If the relevant flags have been set before the optimization function is called, then the code segments are generated in the main program’s source code before the actual optimization commences.

Also, the set of possible values for each parameter has to be specified before invoking the optimization functions. This is explained in detail in chapter ??.

The optimization function follows the *brute force* paradigm: All possible parameter combinations (over defined search spaces) are tried and the performance of their corresponding ASTs is compared before returning the optimal kernel implementation.

First, the 2D-pitched arrays are marked and the corresponding array access macros are defined. Then, the optimizing transformations are successively applied to the AST in the following order:

1. The thread indexing scheme for each *ParFor* loop is checked and changed from *dense* to *sparse* if needed.
2. Any partial accumulation in the `--global--` memory space is staged through an automatic variable.
3. Arrays are cached through the `--shared--` memory spaces if possible.
4. If the flag that controls generation of conditionals to ensure that array indices stay within bounds is set, the relevant conditional statements are generated.

After the application of the above transformations, the AST is traversed and the counter variable of any *serial for* loop is marked. Then, the `.cu` file with the source code of the kernel is generated, compiled along with the main program source code, run, and its execution time is returned. The initial AST parameters and execution time are set as the current best kernel data.

What is left to do now is measure the effect in performance of every AST parameter combination to find the optimal one. The algorithm of going through all possible parameter combinations over the specified sets of values is as follows:

1. Generate a list of all valid double combinations of unroll factors for the *parallel for* loops. For each one, apply the *parallel for* loop unrolling transformation on the AST¹, and go to step 2.
2. Generate a list of all valid values for *serial for* loops for one serial loop counter. For each one, set it as that loop's unroll for, apply the *serial for* loop unrolling transformation on the AST, and go to step 3.
3. If there are still marked loop counters that have not been processed, go to step 2; otherwise, go to step 4.
4. Generate a list of all valid values for the CUDA block width. For each one, set it as the current width, and go to step 5.
5. Generate the kernel source code file, compile and run it. If its performance is better than that of the current best kernel, then the current combination of parameters and execution time is set as the new best one.

After determining the parameters which result in the most efficient kernel, these are restored and the corresponding `.cu` source code file is generated.

Naturally, minor tweaking and optimizing may still be applicable to the kernel (a common one might be removing the possibly unnecessary thread deactivating conditionals), but these should be very easy to handle manually.

¹The ASTs referred to in steps 1 & 2 are the ones in effect when “entering” the corresponding step.

Chapter 3

AutoGPU running instructions

In this chapter, the steps that a user must take to optimize a kernel via the *AutoGPU* program are explained. Execution of the *AutoGPU* program requires a UNIX environment with a Standard ML compiler, a CUDA compiler and a C compiler (required by the CUDA compiler). The program itself was developed and tested on the *Standard ML of New Jersey (SML/NJ)* Standard ML compiler and NVIDIA's *nvcc* CUDA compiler coupled with the GNU *gcc* C compiler.

The *AutoGPU* source code files are listed in section 3.1.

How to specify the input and various parameters of the *AutoGPU* program, as well as how the optimization process is invoked, is explained in section 3.2.

The appropriate execution configuration for the inspected kernel is presented in section 3.3, along with guidelines for developing the necessary “testing platform program”.

3.1 Source code files & compilation

The following files constitute the *AutoGPU* source code package:

<code>dtypes.sml</code>	: Datatype definitions.
<code>sigs.sml</code>	: Signature specifications for each of the defined structures and functors.
<code>kernelargs.sml</code>	: The structure <i>KernelArgs</i> . Contains the tools described in section 2.1.
<code>misc-tools.sml</code>	: The structure <i>MiscTools</i> . Contains the tools described in section 2.2.
<code>alterAST.sml</code>	: The functor <i>AlterAST</i> . Contains the tools described in section 2.3.
<code>translate.sml</code>	: The structure <i>ToCUDA</i> . Contains the tools described in section 2.4.
<code>cmdline.sml</code>	: The functor <i>Cmdline</i> . Contains the tools described in section 2.5.
<code>optim.sml</code>	: The functor <i>Optimizer</i> . Contains the tools described in section 2.6.
<code>functorapp.sml</code>	: Constructs the structures associated with each defined functor, as outlined in figure 2.1. The constructed structures are <i>AltAST</i> , <i>Terminal</i> and <i>Opt</i> .
<code>uses.sml</code>	: A series of <code>use</code> ‘‘...’’ statements, which serves to load and compile the above source code files in the correct order.

To compile the *AutoGPU* program, the user must commence a session with a Standard ML compiler, and issue the command: `use ‘‘uses.sml’’`; The Standard ML environment is then enriched with the *AutoGPU* program, and the user may access its various structures.

3.2 Input, parameters and execution

It has been mentioned in chapter 2 that the various parameters that the *AutoGPU* program needs to optimize a kernel are stored in *mutable* variables. Therefore, there are two ways to assign the desired value to each parameter (including the input AST):

1. Modifying the source code of the *AutoGPU* program. The user must edit the relevant file so that the desired assignment is realized. (All parameters are defined in the beginning of a structure body declaration.)

2. Assigning the desired value to the parameter through a Standard ML compiler environment.

With respect to the second option, the following example may be considered: The user has written the AST of a serial function he or she wishes to optimize, and stored it in file `myast.sml` as `val ast = ParFor (...)`. Then, the interaction with a Standard ML compiler (in this example, the *SML/NJ* compiler) shown in figure 3.1 will assign the user-defined AST to the relevant *AutoGPU* variable (of course, the *AutoGPU* program must be loaded for the following to work):

```
- use "myast.sml";
[opening myast.sml]
val ast =
  ParFor
    ("i", Int 0, Int 1, Sub (Ref #, Int #), true,
     If (Cnd #, Assignment #, Assignment #)) : statement
val it = () : unit
- Opt.ast := ast;
val it = () : unit
-
```

Figure 3.1: Example of specifying a user-defined AST as the *AutoGPU* input program. User input is at lines preceeded with a dash (-).

Of course, it is possible to write the assignment statement `Opt.ast := ast;` in the prepared file; thus, loading the file will also assign the desired value to the relevant *AutoGPU* parameter. Furthermore, the prepared file need not contain only one value declaration and/or assignment: The user may prepare a file that takes care of all *AutoGPU* parameters.

The parameter declarations are divided among the source code files, however all can be accessed through the highest-order structure, *Opt*. Hence, any parameter `x` may be accessed as `Opt.x`.

Kernel arguments

The kernel parameter variables are declared in file `kernelargs.sml` and are the following:

- `programe` is the string of the desired kernel name.
- `kargs` is the list of the formal kernel arguments.
- `constants` is the list of constants available to the kernel.
- `macroarrays` is a list of all the 2D-pitched arrays referenced in the kernel body.
- `arraybounds` is a list containing the bounds variables of specified references (normally arrays, but the user may define any variable to be “bounded”).

To clarify, an example of a kernel declaration along with the constants it operates upon, and the corresponding *AutoGPU* parameter assignments is presented in figure 3.2.

There are three points that should be noted, regarding the example of figure 3.2:

- The `macroarrays` list need not be defined by the user, as this is done automatically before the optimization process commences.
- As has been mentioned in section 2.1, the array pitches for any 2D arrays are also automatically appended in the `constants` lists.
- The `arraybounds` list needs to be defined only if the user intends to have the *AutoGPU* program generate conditionals to control accesses to the variables this list contains.

```

__constant__ int M;
__constant__ int N;

// array2 is an MxN matrix
__global__ void myKernel( float* array2, int threshold )
{
    // ... the kernel's body ...
}

Opt.progname      := "myKernel";
Opt.kargs         := [ ("array2", POINTER REAL), ("threshold", INT) ];
Opt.constants     := [ ("M", INT), ("N", INT) ];
Opt.arraybounds   := [ ("array2", [ "M", "N" ]) ]

```

Figure 3.2: Example of specifying a kernel's arguments and constants in the *AutoGPU* program

Loop parameters

The parameters that control the parallel loop block sizes and counter declarations, as well as the unrolling of all loops are declared in file `misc-tools`. These are the following:

- **ndims** is the kernel program domain dimension, or the number of dimensions of the arrays that the kernel operates on.
- **width** is the CUDA block width, or the number of threads per CUDA block.
- **unrlPar** is a 2-tuple that contains the unroll factors of the parallel loops. (If **ndims** has a value of 1, then only the first element of **unrlPar** is referenced, but it still needs to be a 2-tuple.)
- **unrl** is a list of the serial loop counters and their respective unroll factors.

All of the above parameters are assigned values by the *AutoGPU* program as part of the optimization process. However, if the user wishes that the *AutoGPU* program simply generate a specific version of a kernel (see section 2.6), he or she may assign the desired values to these parameters and call function `getCurrKernel` (see also *Kernel generation*).

An example of specifying a set of loop parameters in the *AutoGPU* program is presented in figure 3.3.

Block width	Domain dimension	Loop unroll factors			
		Parallel loops		Serial loops	
		<i>i</i> loop	<i>j</i> loop	<i>k</i> loop	<i>l</i> loop
128	2	2	4	5	10

```

Opt.ndims      := 2;
Opt.width      := 128;
Opt.unrlPar    := (2, 4);
Opt.unrl       := [ ("k", 5), ("l", 10) ];

```

Figure 3.3: Example of specifying a desired loop configuration in the *AutoGPU* program

Floating-point number precision

Floating-point number precision is controlled by variable `prec`, declared in file `translate.sml`. As stated in subsection 1.2.2, it may be assigned any of the following values:

- FLOAT
- DOUBLE
- LONGDOUBLE

The default precision is that of the *float* type, and it controls the precision of all floating-point variables or literals referenced in the kernel.

Compiled program configuration

The *AutoGPU* optimization process requires that the compiled program be executed, so that its execution time may be measured. As a result, the following parameters, declared in file `cmdline.sml`, must be defined:

- `execArgs`, which is a list that contains all command-line arguments that must be passed to the compiled program.
- `execLoc`, which is a string representation of the directory where the makefile target (i.e. the compiled program) is created. This may either be a relative or an absolute UNIX path.

For a program that is compiled by a makefile which follows the CUDA SDK model and which receives two command-line arguments, the appropriate parameter assignments are shown in figure 3.4.

```
Opt.execArgs := ["data1", "data2"];
Opt.execLoc  := "/home/user/NVIDIA_GPU_Computing_SDK/C/bin/linux/release";
```

Figure 3.4: Example of specifying the execution configuration for a program with 2 command-line arguments, compiled in the CUDA SDK environment.

Kernel generation

If the user wishes to obtain a kernel described by a set of parameters he or she has already specified in the *AutoGPU* program, then the relevant function has to be invoked. This is achieved by the command:

```
Opt.getCurrKernel ();
```

The kernel source code is generated in file `kernel-progname.cu`, stored in the current directory.

If, on the other hand, the user wishes to obtain the optimal kernel as it results by the *AutoGPU* optimization process, certain parameters have to be defined first. All of those are declared in file `optim.sml`.

Three boolean flag parameters are declared:

- `genConds` controls whether the conditional-generating transformation described in subsection 2.3.5 will be applied.
- `genTimer` controls whether the code segments which carry out the necessary timing process in the main program will be generated in the main program source code (see appendix A for the relevant source code).
- `genErrChk` controls whether the code segments which perform checking if the kernel failed to launch will be generated in the main program source code (see appendix A for the relevant source code).

The rest of the parameters are used to create the sets of possible values for the `width`, `unr1Par` and `unr1` parameters. Generation of the set for each parameter element is controlled by three parameters, and the resulting set of size N for each one is the following:

$$possible_values_set = \{l, f(l, b), f(f(l, b), b), \dots, f(f(\dots f(l, b), \dots), b)\}$$

The N , l , f and b values are defined in the following parameters:

- `xxxxL` is the lowest value l that the parameter `xxxx` may take.
- `xxxxN` is the number of different values N that will be tried for parameter `xxxx` during the optimization process.
- `xxxxS` is a 2-tuple containing a function f on two integers and an integer value, that is used as the second operand b of function f .

Two examples of how the above parameters may be used to generate a desired set are presented in figures 3.5 and 3.6.

$$target_set = \{1, 2, 4, 8, 16\}$$

<pre>xxxxL := 1; xxxxN := 5; xxxxS := (op*, 2);</pre>

Figure 3.5: Example of how to specify generation of a set of powers of 2 in the *AutoGPU* program

$$target_set = \{1, 2, 3, 4, 6, 12\}$$

<pre>local val i2r = real.fromInt in fun divisors (prevd, num) = let val d = prevd + 1 in if ((i2r num) / (i2r d)) = i2r (num div d) then d else divisors (d, num) end end; xxxxL := 1; xxxxN := 6; xxxxS := (divisors, 12);</pre>

Figure 3.6: Example of how to specify generation of a set of divisors of 12 in the *AutoGPU* program

If all necessary parameters of the *AutoGPU* program have been set, the optimization process is invoked with the command:

```
Opt.optimize ();
```

Naturally, the optimization process execution time will vary, depending on the computational load of the kernel, the size of the testing input data, and the number of loop parameter combinations. As a reference, the optimization of the kernels presented in chapter ?? required the *AutoGPU* program to run for approximately 2 hours.

3.3 Testing platform development

Each version of a kernel that is generated by the *AutoGPU* program must be run, in order to measure the execution time of each successive kernel version, which provides the mechanism to deduce which version is the optimal one. Nonetheless, a CUDA kernel cannot execute on its own: A main program is needed.

This main program must read the kernel’s input data, allocate space in GPU memory, copy data to it, set the CUDA kernel execution configuration, and free allocated memory. All these, plus copying the kernel’s result back to host memory, comprise a typical main program of a CUDA kernel. However, a main program developed to invoke a kernel being optimized by the *AutoGPU* program must follow a specific model. Such a main program is referred to as the “testing platform” of the kernel.

Two examples of a testing platform’s source code can be found in appendix A. The presented testing platforms are the ones used for the optimization of the kernels discussed in chapter 4.

Including the kernel file

The kernel source code is generated in a separate file than the testing platform source code, and the file of the generated kernel contains certain macro definitions that must be loaded in the scope of the testing platform as well. As a result, the kernel source code must be `#included` in the testing platform source code, with the following directive to the CUDA preprocessor:

```
#include ‘‘kernel-progname.cu’’
```

The `#include` directive must specify the kernel source code file with double quotes rather than angled brackets, because the kernel file is generated in the same directory as the testing platform file, instead of the *library* directory maintained by the compiler preprocessor.

Main program input

The *AutoGPU* program was developed to be able to optimize (or at least operate on) any kernel. Consequently, automatic generation of input data for a kernel is not possible; the user must supply the data the kernel will operate on. Naturally, the supplied input should be a set of data that is representative of the input that the optimal kernel is expected to receive. Care must be taken when considering the input’s volume: As the input becomes larger in size, the optimization process will need to execute for a longer period of time; on the other hand, too small an input may result in kernel that is not optimal for a specific input size and beyond. Hence, a golden mean must be found, as far as the size of the input is concerned.

Furthermore, since the kernel optimization process is automatic, the main program may not receive input via the terminal during execution. Instead, the program’s input must be stored in files and passed to it as command-line arguments.

CUDA execution configuration

The CUDA block and grid dimensions depend on the parallel loop parameters of the kernel. These are defined in macros during generation of the kernel’s source code, and should be used to specify the CUDA execution configuration of the kernel.

Any 2-dimensional matrix is presupposed to be stored in column-major fashion, hence the appropriate CUDA grid and block dimensions are as shown in table 3.1. Variables in capital letters correspond to the automatically generated macros.

Output size	Grid size	Block size
m	$\lceil \frac{m}{WIDTH \cdot UNROLL1} \rceil$	$WIDTH$
$m \times n$	$\lceil \frac{m}{WIDTH \cdot UNROLL1} \rceil \times \lceil \frac{n}{UNROLL2} \rceil$	$WIDTH$

Table 3.1: Appropriate CUDA execution configuration for 1D & 2D kernels

GPU memory data

Memory allocation and assignment for all data that must reside in GPU memory must be performed by the testing platform before the kernel's invocation. The only exception is the case of `__constant__` variables which are defined in the corresponding *AutoGPU* parameter (see section 3.2), including pitches for 2D arrays; these constants need only have their value assigned by the testing platform, since their declarations are generated in the kernel source code file.

Kernel execution time measurement

It is impossible to properly measure the execution time of a CUDA kernel from the scope of the *AutoGPU* program; this must be realized by the testing platform. In accordance with the *AutoGPU* model, the testing platform must store the measured execution time in a file `comptime.agpu`, in the same directory where the testing platform, kernel, and input data are stored.

Kernel launch failure checking

A kernel generated during the optimization process may fail to launch. This might happen because the size of the CUDA blocks exceeds the maximum size allowed by the resident GPU, or because the number of registers that have to be allocated to each block exceeds the number of registers available to a multiprocessor, and so forth.

In this case, the main program will execute normally, yet the measured execution time will be no more than several microseconds. This measurement must be discarded, otherwise the faulty kernel will be set as the optimal one, due to its seemingly low execution time. Therefore, the testing platform should check for CUDA runtime errors after the kernel function has returned, and return an integer value other than 0 at its termination. This will be interpreted by the *AutoGPU* program as an execution time of `MaxInt` seconds, effectively discarding the faulty kernel.

Chapter 4

Examples

In this chapter, the *AutoGPU* program is applied to the AST representations of the serial implementations of two basic computational operations, in order to demonstrate the results that can be yielded by it. Application of the program will automatically produce the final optimized kernel, but here also intermediate versions will be shown along with performance statistics, so that the optimizational steps may be outlined clearly. In each example, the kernel suffixed with `_autogpu` is the one that is automatically produced by application of the *AutoGPU* program.

All reported execution statistics refer only to the execution of the kernel in the GPU, and not that of the whole (i.e. host-to-host) execution.

All matrices that are supplied to the kernel as input throughout this chapter refer to square matrices with random floating-point (single precision) elements in the interval $[-1.0, +1.0]$, generated with the use of the C standard library function `rand()`.

Reported kernel execution statistics refer only to the execution of the kernel in the GPU, and not that of the whole (i.e. host-to-host) execution. These statistics were measured on the *quattro* server of the Electrical & Computer Engineering Department at the Aristotle University of Thessaloniki, School of Engineering. The server's computation units are shown in table 4.1.

GPU	NVIDIA Tesla C1060 Clock rate @ 1.3GHz 240 cores, 30 multiprocessors
CPU	Intel ...something...

Table 4.1: The computation units of AUTH' *quattro* server

The output of every reported kernel has been checked for correctness against the output of its serial version. Of course, some error is expected, due to differences between the GPU & CPU computation models (e.g. rounding policy). The maximum absolute error of the optimized parallel version will be reported for certain input sizes as a point of reference. It should be mentioned that different parallel versions of the same original (serial) AST produce identical output.

As explained in section 2.6, different widths for the CUDA blocks are tried along with the various loop unrolling factors. As a result, the *AutoGPU* program does not provide any information as to the optimal CUDA block width of the intermediate kernels that will be presented throughout this chapter: These kernels are arbitrarily defined to execute in CUDA blocks of 128 threads each.

4.1 Matrix Multiplication: MATMUL

Matrix multiplication is a typical example of a computational process that can be easily implemented by a CUDA kernel. The algorithm is characteristically simple and operation-intensive thus allowing a clearly displayed exploitation of the CUDA programming model. The methodology for manually performing the following optimizations is explained in detail in [3]. Here, the same

(or corresponding) successive versions of the kernel are produced automatically by the *AutoGPU* program. The performance of the final optimized kernels will then be compared.

The computed product is that of the expression $A = B \cdot C$. Matrices are stored in column-major fashion and their dimension variables are as follows:

$$A : m \times n \quad , \quad B : m \times p \quad , \quad C : p \times n$$

Two cases of matrix multiplication will be studied: One where the input matrix sizes are expected to be multiples of the size that each CUDA block operates upon; and one where the input matrices may be of any (valid) size. The latter requires the generation of the conditional statements described in subsection 2.3.5, whereas the former does not.

The code for the simple serial implementation of matrix multiplication is shown in figure 4.1.

```
void matmul_serial( float* A, float* B, float* C, int m, int n, int p )
{
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < p; ++k)
                A[i + j*m] += B[i + k*m] * C[k + j*p];
}
```

Figure 4.1: mm_s: Serial MATMUL kernel

4.1.1 Without conditionals

In this subsection, five parallel implementations of matrix multiplication will be compared:

- mm_0 : The “vanilla” parallel kernel—a direct translation of the input AST to CUDA source code.
- mm_1 : The parallel kernel, after staging of partial accumulations.
- mm_2 : The parallel kernel, after caching of array references.
- mm_autogpu : The optimized parallel kernel, after unrolling of both parallel and serial loops.
- mm_wolfe : Michael Wolfe’s k4-128y2x4x4.cu kernel.

Successive versions of the kernel

First, the AST must be scanned for 2D-pitched array references, in order to deduce the appropriate macros and `__constant__` pitches. Afterwards, the vanilla parallel version of the MATMUL kernel can be obtained, whose CUDA code is shown in figure 4.2.

Macros:

`X_at(i,j) ≡ (*(float*)((char*)X + j*pitch_X) + i)`

`WIDTH ≡ 128` `UNROLL1 ≡ 1` `UNROLL2 ≡ 1`

```
__global__ void matmul( float* A, float* B, float* C, int m, int n, int p )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    if (((1 - 0)%1) || (i < 0) || (i > (m - 1)))
        return;
    int j = blockIdx.y*UNROLL2;
    if (((j - 0)%1) || (j < 0) || (j > (n - 1)))
        return;

    for (int k = 0; k <= (p - 1); k += 1) {
        A_at(i,j) += (B_at(i,k) * C_at(k,j));
    } // end for (counter k)
}
```

Figure 4.2: mm_0: Vanilla parallel MATMUL kernel

Naturally, 3 array macros are generated, one for each 2D-array. Also, the *if* statements which deal with deactivating redundant threads are redundant in this (and all subsequent) kernel. This is because the *ParFor* statement which generated them is parametrized with *first* = 0, *last* = *m* (or *n*), *stride* = 1, so that no threads need to be deactivated. These statements will be omitted from now on.

The first optimizing transformation can now be applied. The code in figure 4.3 results from staging the partial accumulation of *A.at(i,j)*.

Macros:

X.at(i,j) \equiv $(*(\text{float}*)(\text{char}*)X + j*\text{pitch}_X) + i$

WIDTH \equiv 128 *UNROLL1* \equiv 1 *UNROLL2* \equiv 1

```
__global__ void matmul( float* A, float* B, float* C, int m, int n, int p )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;

    float acc1;
    acc1 = A_at(i,j);

    for (int k = 0; k <= (p - 1); k += 1) {
        acc1 += (B_at(i,k) * C_at(k,j));
    } // end for (counter k)

    A_at(i,j) = acc1;
}
```

Figure 4.3: mm_1: Staged accumulation parallel MATMUL kernel

In version mm_1, all threads of the same block access the same element of matrix *C*, a behavior which can be optimized by caching the *C* array through *__shared__* memory. The resulting code is shown in figure 4.4

Macros:

X.at(i,j) \equiv $(*(\text{float}*)(\text{char}*)X + j*\text{pitch}_X) + i$

WIDTH \equiv 128 *UNROLL1* \equiv 1 *UNROLL2* \equiv 1

```
__global__ void matmul( float* A, float* B, float* C, int m, int n, int p )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;
    int tx = threadIdx.x;

    __shared__ float C_c[WIDTH];
    float acc1;
    acc1 = A_at(i,j);

    for (int k_s = 0; k_s <= (p - 1); k_s += (1 * WIDTH)) {
        __syncthreads();
        C_c[tx] = C_at((k_s+tx),j);
        __syncthreads();

        for (int k = k_s; k <= ((k_s + WIDTH) - 1); k += 1) {
            acc1 += (B_at(i,k) * C_c[(k - k_s)]);
        } // end for (counter k)
    } // end for (counter k_s)

    A_at(i,j) = acc1;
}
```

Figure 4.4: mm_2: Staged & cached parallel MATMUL kernel

The kernel can be further optimized by unrolling the *i*, *j* and *k* loops. All combinations of loop unroll factors in the set {1,2,4,8,16} are tried and the kernel with the best performance is returned. Its code is shown in figure 4.5.

Unrolling of the loops causes an increase in the source code's volume, which may heavily affect readability. The code presented in figure 4.5 is the automatically generated one, after some “tidying up” has been done, including issuing the `#pragma unroll` directive before the k loop instead of unrolling it explicitly.

Macros:

$X_at(i,j) \equiv (*(float*)(char*)X + j*pitch_X) + i$

$WIDTH \equiv 256$ $UNROLL1 \equiv 2$ $UNROLL2 \equiv 8$

```
__global__ void matmul( float* A, float* B, float* C, int m, int n, int p )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;
    int tx = threadIdx.x;

    __shared__ float C_c_1[WIDTH], __shared__ float C_c_2[WIDTH];
    __shared__ float C_c_3[WIDTH], __shared__ float C_c_4[WIDTH];
    __shared__ float C_c_5[WIDTH], __shared__ float C_c_6[WIDTH];
    __shared__ float C_c_7[WIDTH], __shared__ float C_c_8[WIDTH];

    float acc1_1_1 = A_at(i,j);
    float acc1_1_2 = A_at(i, (j + 1));
    float acc1_1_3 = A_at(i, (j + 2));
    float acc1_1_4 = A_at(i, (j + 3));
    float acc1_1_5 = A_at(i, (j + 4));
    float acc1_1_6 = A_at(i, (j + 5));
    float acc1_1_7 = A_at(i, (j + 6));
    float acc1_1_8 = A_at(i, (j + 7));
    float acc1_2_1 = A_at((i + (1 * WIDTH)),j);
    float acc1_2_2 = A_at((i + (1 * WIDTH)), (j + 1));
    float acc1_2_3 = A_at((i + (1 * WIDTH)), (j + 2));
    float acc1_2_4 = A_at((i + (1 * WIDTH)), (j + 3));
    float acc1_2_5 = A_at((i + (1 * WIDTH)), (j + 4));
    float acc1_2_6 = A_at((i + (1 * WIDTH)), (j + 5));
    float acc1_2_7 = A_at((i + (1 * WIDTH)), (j + 6));
    float acc1_2_8 = A_at((i + (1 * WIDTH)), (j + 7));

    for (int k_s = 0; k_s <= (p - 1); k_s += (1 * WIDTH)) {
        __syncthreads();
        C_c_1[tx] = C_at((k_s+tx),j);
        C_c_2[tx] = C_at((k_s+tx), (j + 1));
        C_c_3[tx] = C_at((k_s+tx), (j + 2));
        C_c_4[tx] = C_at((k_s+tx), (j + 3));
        C_c_5[tx] = C_at((k_s+tx), (j + 4));
        C_c_6[tx] = C_at((k_s+tx), (j + 5));
        C_c_7[tx] = C_at((k_s+tx), (j + 6));
        C_c_8[tx] = C_at((k_s+tx), (j + 7));
        __syncthreads();

        #pragma unroll 16
        for (int k = k_s; k <= ((k_s + WIDTH) - 1); k += 1) {
            acc1_1_1 += (B_at(i,k) * C_c_1[(k - k_s)]);
            acc1_1_2 += (B_at(i,k) * C_c_2[(k - k_s)]);
            acc1_1_3 += (B_at(i,k) * C_c_3[(k - k_s)]);
            acc1_1_4 += (B_at(i,k) * C_c_4[(k - k_s)]);
            acc1_1_5 += (B_at(i,k) * C_c_5[(k - k_s)]);
            acc1_1_6 += (B_at(i,k) * C_c_6[(k - k_s)]);
            acc1_1_7 += (B_at(i,k) * C_c_7[(k - k_s)]);
            acc1_1_8 += (B_at(i,k) * C_c_8[(k - k_s)]);
            acc1_2_1 += (B_at((i + (1 * WIDTH)),k) * C_c_1[(k - k_s)]);
            acc1_2_2 += (B_at((i + (1 * WIDTH)),k) * C_c_2[(k - k_s)]);
            acc1_2_3 += (B_at((i + (1 * WIDTH)),k) * C_c_3[(k - k_s)]);
            acc1_2_4 += (B_at((i + (1 * WIDTH)),k) * C_c_4[(k - k_s)]);
            acc1_2_5 += (B_at((i + (1 * WIDTH)),k) * C_c_5[(k - k_s)]);
            acc1_2_6 += (B_at((i + (1 * WIDTH)),k) * C_c_6[(k - k_s)]);
            acc1_2_7 += (B_at((i + (1 * WIDTH)),k) * C_c_7[(k - k_s)]);
            acc1_2_8 += (B_at((i + (1 * WIDTH)),k) * C_c_8[(k - k_s)]);
        } // end for (counter k)
    } // end for (counter k_s)
}
```

```

A_at(i,j) = acc1_1_1;
A_at(i,(j + 1)) = acc1_1_2;
A_at(i,(j + 2)) = acc1_1_3;
A_at(i,(j + 3)) = acc1_1_4;
A_at(i,(j + 4)) = acc1_1_5;
A_at(i,(j + 5)) = acc1_1_6;
A_at(i,(j + 6)) = acc1_1_7;
A_at(i,(j + 7)) = acc1_1_8;
A_at((i + (1 * WIDTH)),j) = acc1_2_1;
A_at((i + (1 * WIDTH)),(j + 1)) = acc1_2_2;
A_at((i + (1 * WIDTH)),(j + 2)) = acc1_2_3;
A_at((i + (1 * WIDTH)),(j + 3)) = acc1_2_4;
A_at((i + (1 * WIDTH)),(j + 4)) = acc1_2_5;
A_at((i + (1 * WIDTH)),(j + 5)) = acc1_2_6;
A_at((i + (1 * WIDTH)),(j + 6)) = acc1_2_7;
A_at((i + (1 * WIDTH)),(j + 7)) = acc1_2_8;
}

```

Figure 4.5: `mm_autogpu`: Optimal MATMUL kernel

The kernel execution time (figure 4.6) and performance in GFLOPS (figure 4.7) are shown for the serial and all parallel implementations of matrix multiplication, tested on a series of input matrices of increasing sizes. Hence, the benefit of each optimizing transformation can be graphically displayed. The serial kernel’s performance is shown only for a subset of the input matrices, as it is dramatically lower than that of the parallel kernels.

Reference kernel: `mm_wolfe`

To help appreciate the result that can be achieved by the *AutoGPU* program, its output is compared to the `mm_wolfe` reference kernel. The reference kernel’s source code is shown in figure 4.8.

As can be seen, the `mm_autogpu` and `mm_wolfe` kernels are very similar, since the same optimizations have been applied to each. The difference lies in the defined loop configurations and unroll factors, which may be seen at table 4.2. This is expected, since the loop parameters set by the *AutoGPU* program is decided upon as the optimal one (for the GPU it was tried on) between 625 different sets—needless to say, it is realistically impossible for all these combinations to be tried manually.

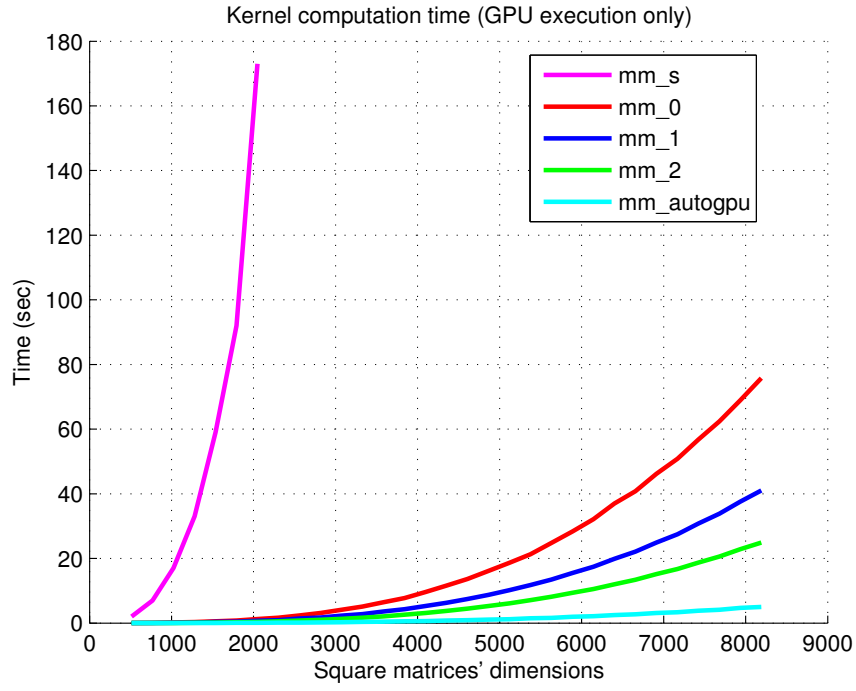


Figure 4.6: Execution times of the input, intermediate and final MATMUL kernels

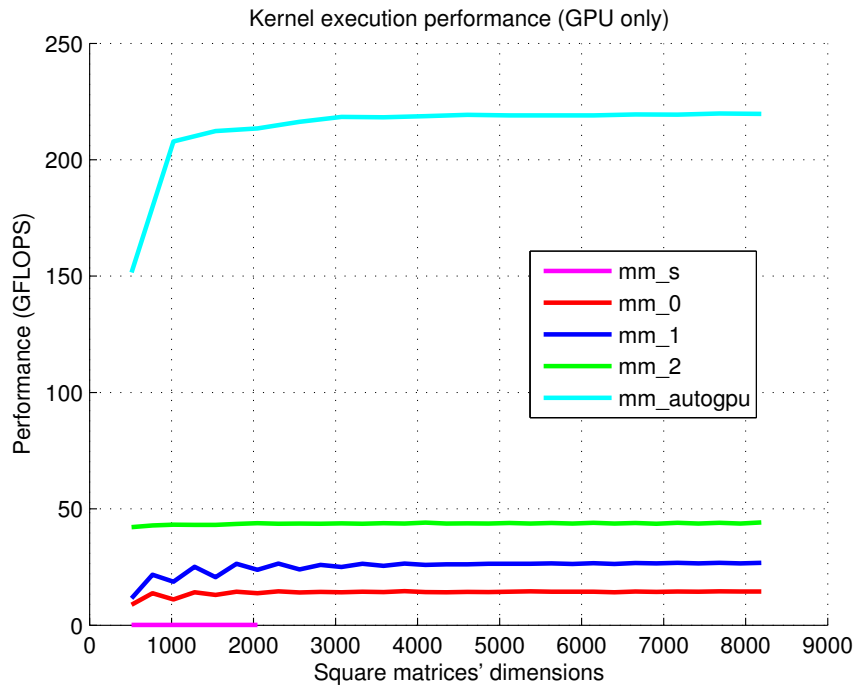


Figure 4.7: Performance of the input, intermediate and final MATMUL kernels

```

__global__ void matmul( float* a, float* b, float* c,
                        int pitch_a, int pitch_b, int pitch_c,
                        int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*256 + tx;
    int j = blockIdx.y*4;
    __shared__ float cb0[256], cb1[256], cb2[256], cb3[256];

    float sum00 = 0.0f, sum01 = 0.0f, sum02 = 0.0f, sum03 = 0.0f;
    float sum10 = 0.0f, sum11 = 0.0f, sum12 = 0.0f, sum13 = 0.0f;
    for( int ks = 0; ks < p; ks += 128 ){
        cb0[tx] = c[ks+tx+pitch_c*j];
        cb1[tx] = c[ks+tx+pitch_c*(j+1)];
        cb2[tx] = c[ks+tx+pitch_c*(j+2)];
        cb3[tx] = c[ks+tx+pitch_c*(j+3)];
        __syncthreads();
        for( int k = 0; k < 128; k+=4 ){
            float rb = b[i+pitch_b*(k+ks)];
            sum00 += rb * cb0[k];
            sum01 += rb * cb1[k];
            sum02 += rb * cb2[k];
            sum03 += rb * cb3[k];
            rb = b[i+pitch_b*(k+ks+1)];
            sum00 += rb * cb0[k+1];
            sum01 += rb * cb1[k+1];
            sum02 += rb * cb2[k+1];
            sum03 += rb * cb3[k+1];
            rb = b[i+pitch_b*(k+ks+2)];
            sum00 += rb * cb0[k+2];
            sum01 += rb * cb1[k+2];
            sum02 += rb * cb2[k+2];
            sum03 += rb * cb3[k+2];
            rb = b[i+pitch_b*(k+ks+3)];
            sum00 += rb * cb0[k+3];
            sum01 += rb * cb1[k+3];
            sum02 += rb * cb2[k+3];
            sum03 += rb * cb3[k+3];
            rb = b[i+128+pitch_b*(k+ks)];
            sum10 += rb * cb0[k];
            sum11 += rb * cb1[k];
            sum12 += rb * cb2[k];
            sum13 += rb * cb3[k];
            rb = b[i+128+pitch_b*(k+ks+1)];
            sum10 += rb * cb0[k+1];
            sum11 += rb * cb1[k+1];
            sum12 += rb * cb2[k+1];
            sum13 += rb * cb3[k+1];
            rb = b[i+128+pitch_b*(k+ks+2)];
            sum10 += rb * cb0[k+2];
            sum11 += rb * cb1[k+2];
            sum12 += rb * cb2[k+2];
            sum13 += rb * cb3[k+2];
            rb = b[i+128+pitch_b*(k+ks+3)];
            sum10 += rb * cb0[k+3];
            sum11 += rb * cb1[k+3];
            sum12 += rb * cb2[k+3];
            sum13 += rb * cb3[k+3];
        }
        __syncthreads();
    }
}

```

```

a[i+pitch_a*j] = sum00;
a[i+pitch_a*(j+1)] = sum01;
a[i+pitch_a*(j+2)] = sum02;
a[i+pitch_a*(j+3)] = sum03;
a[i+128+pitch_a*j] = sum10;
a[i+128+pitch_a*(j+1)] = sum11;
a[i+128+pitch_a*(j+2)] = sum12;
a[i+128+pitch_a*(j+3)] = sum13;
}

```

Figure 4.8: `mm_wolfe`: Michael Wolfe’s kernel

The optimizations that have been applied to the optimal generated kernel and the reference one, as well as the loop parameters for each, are summarized in table 4.2.

Kernel	Block width	Staging	__shared__ caching	Loop unroll factors		
				<i>i</i> loop	<i>j</i> loop	<i>k</i> loop
<code>mm_autogpu</code>	256	Yes	Yes	2	8	16
<code>mm_wolfe</code>	128	Yes	Yes	2	4	4

Table 4.2: Optimal and reference `MATMUL` kernels specifications

The performance statistics obtained by each of the two kernels, for input matrices of sizes 512×512 to 16896×16896 , are shown in figures 4.9 and 4.10.

The advantage of being able to determine the best set of unroll factors (within a defined search space) is apparent: The `mm_autogpu` kernel reaches a performance of 220 GFLOPS, while the `mm_wolfe` kernel stops at 175 GFLOPS.

Kernel precision

Table 4.3 lists the maximum absolute error in the product matrix computed in the GPU (as compared to that of the CPU), for different input matrix sizes.

Matrix size	Maximum absolute error
512×512	$1.14441 \cdot 10^{-5}$
1024×1024	$2.28882 \cdot 10^{-5}$
2048×2048	$5.34058 \cdot 10^{-5}$
4096×4096	$7.62939 \cdot 10^{-5}$

Table 4.3: `MATMUL` kernel (no conditionals) output’s maximum absolute error

It should be kept in mind that the kernels discussed in this subsection can only operate on matrices whose sizes are in accordance with the CUDA execution configuration for each one. For example, in the case of the `mm_autogpu` kernel, this means that input matrices should adhere to the following specification: $\mathbf{B} : (512 \cdot a) \times (16 \cdot b)$, $\mathbf{C} : (16 \cdot b) \times (8 \cdot c)$, where $512 = \text{WIDTH} \cdot \text{unroll}_i\text{_loop}$, $16 = \text{unroll}_k\text{_loop}$ and $8 = \text{unroll}_j\text{_loop}$.

Conclusions

In tables 4.4 and 4.5 a quantitative comparison of the discussed kernels is presented for input matrices of size 2048×2048 , regarding execution time and floating-point operations, respectively.

These statistics (both graphical and numerical) indicate the importance of each optimization. First of all, staging the accumulation of every element of the product matrix **A** (a very simple process for the `MATMUL` kernel) results in an almost twice as fast kernel as the direct translation of the serial one. The same happens when stripes of of input matrix **C** are cached through the `__shared__` memory space. This is because *warp size* elements of the **C** matrix are processed in a

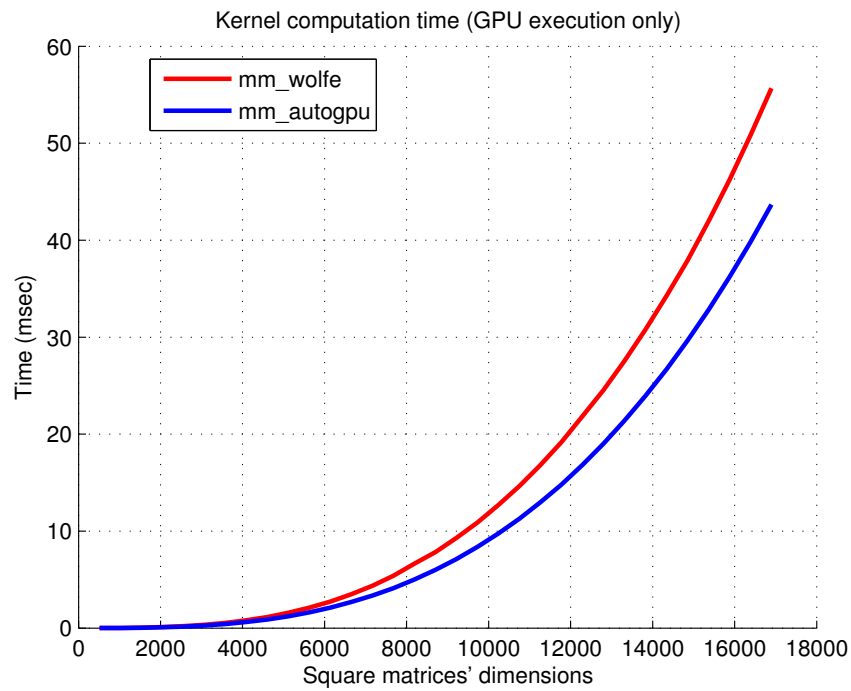


Figure 4.9: Execution times of the reference and optimal MATMUL kernels

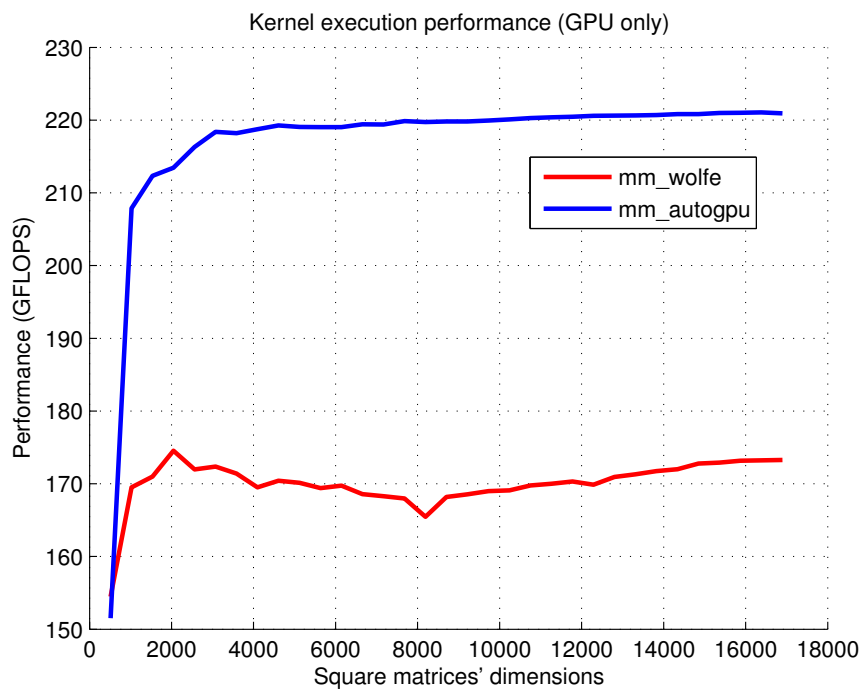


Figure 4.10: Performance of the reference and optimal MATMUL kernels

Kernel	Execution time	Approximate speed-up			
		to mm_0	to mm_1	to mm_2	to mm_wolfe
mm_0	1248 msec	–	–	–	–
mm_1	727 msec	42%	–	–	–
mm_2	392 msec	69%	46%	–	–
mm_wolfe	98 msec	92%	87%	75%	–
mm_autogpu	72 msec	94%	90%	81%	27%

Table 4.4: MATMUL kernel execution times and relative speed-up for 2048×2048 matrices

Kernel	Performance	Approximate speed-up			
		to mm_0	to mm_1	to mm_2	to mm_wolfe
mm_0	14 GFLOPS	–	–	–	–
mm_1	24 GFLOPS	71%	–	–	–
mm_2	44 GFLOPS	214%	83%	–	–
mm_wolfe	175 GFLOPS	1150%	629%	298%	–
mm_autogpu	235 GFLOPS	1579%	879%	434%	34%

Table 4.5: MATMUL kernel FLOPS and relative speed-up for 2048×2048 matrices

single memory operation in the body of the k -s loop (instead of *warp size* memory operations), and in the body of the k loop, each element of matrix **C** is accessed simultaneously by all threads of the warp, as fast as a register would be.

However, this is not enough: GPUs are able to perform calculations and memory transfers concurrently, therefore the memory latency may be hidden by a massive computation load. This may be realized by having each thread compute the value of more than 1 product matrix element, i.e. by unrolling the parallel loops. Of course, there are limits as to how much the parallel loops can be unrolled, as explained in subsection 2.3.4.

Owing to this reason, as well as because it generally decreases the overhead for each loop, loop unrolling results in a dramatic performance boost, especially with respect to the FLOPS achieved by the kernel. Also, it is apparent by the gain achieved by the `mm_autogpu` kernel, with regard to the otherwise very similar `mm_wolfe` kernel, that determining the loop parameters (unroll factors as well as block size) is rather important for exploiting the capabilities of a GPU.

4.1.2 With conditionals

In this subsection, five parallel implementations of matrix multiplication, analogous to the ones discussed in subsection 4.1.1, will be compared:

- `mm_cond_0` : A direct translation of the input AST to CUDA source code, enhanced with conditionals to ensure that no out-of-bounds array accesses will occur.
- `mm_cond_1` : The parallel kernel after staging of partial accumulations.
- `mm_cond_2` : The parallel kernel, after caching of array references.
- `mm_cond_autogpu` : The optimized parallel kernel, after unrolling of both parallel and serial loops.
- `mm_cond_wolfe` : Michael Wolfe’s `k4safe-128x4x4.cu` kernel.

It should be noted that during the kernel optimization process –as implemented in the *AutoGPU* program– the AST transformation to add conditionals to stay within array bounds is applied only before the loops are unrolled. This is because nodes where array references occur may differ between the input AST and the one resulting from the staging or caching transformations. Here, the conditionals are inserted in the kernel all intermediate transformations have been applied, for each kernel.

Successive versions of the kernel

Because in the following versions of the MATMUL kernel the matrix dimensions will be frequently referenced, these variables will be defined as the `__constants__` M, N and P.

Apart from the generation of conditional statements to control matrix element access, the optimizing transformations have exactly the same effect on the kernel's AST, hence the effect of the applied transformations will not be further commented upon.

Figure 4.11 shows the straightforward implementation of the MATMUL kernel, with conditionals added.

Macros:

`X_at(i,j) ≡ (*(float*)((char*)X + j*pitch_X) + i))`

`WIDTH ≡ 128 UNROLL1 ≡ 1 UNROLL2 ≡ 1`

```
__global__ void matmul( float* A, float* B, float* C )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;

    for (int k = 0; k <= (P - 1); k += 1) {
        if (i < M) {
            if (j < N) {
                A_at(i,j) += (B_at(i,k) * C_at(k,j));
            } else {
            } // end if-else
        } else {
        } // end if-else
    } // end for (counter k)
}
```

Figure 4.11: `mm_cond_0`: Vanilla parallel MATMUL kernel (any matrix size)

The generation of *if* statements without an *else* part, allowed by the *Blank* statement as mentioned in subsection 1.2.4 can be seen in figure 4.11. These do not affect performance negatively, as they are taken care of by the `nvcc` compiler. Since they are redundant and make the kernels more difficult to read, they will be omitted from now on.

Figures 4.12 and 4.13 show the kernels obtained after the staging and caching transformations have been applied, respectively.

The kernel that is ultimately produced by the *AutoGPU* program is shown in figure 4.14.

Macros:

X_at(i,j) \equiv `(*(float*)(char*)X + j*pitch_X) + i`)

WIDTH \equiv 128 **UNROLL1** \equiv 1 **UNROLL2** \equiv 1

```
__global__ void matmul( float* A, float* B, float* C )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;

    float accl;
    if (i < M) {
        if (j < N) {
            accl = A_at(i,j);
        } // end if-else
    } // end if-else

    for (int k = 0; k <= (P - 1); k += 1) {
        if (i < M) {
            if (k < P) {
                accl += (B_at(i,k) * C_at(k,j));
            } // end if-else
        } // end if-else
    } // end for (counter k)

    if (i < M) {
        if (j < N) {
            A_at(i,j) = accl;
        } // end if-else
    } // end if-else
}
```

Figure 4.12: `mm_cond_1`: Staged accumulation parallel MATMUL kernel (any matrix size)

Macros:

$X_at(i,j) \equiv (*(float*)(char*)X + j*pitch_X) + i$

$WIDTH \equiv 128$ $UNROLL1 \equiv 1$ $UNROLL2 \equiv 1$

```
__global__ void matmul( float* A, float* B, float* C )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;
    int tx = threadIdx.x;

    __shared__ float C_c[WIDTH];
    float acc1;
    if (i < M) {
        if (j < N) {
            acc1 = A_at(i,j);
        } // end if-else
    } // end if-else

    for (int k_s = 0; k_s <= (P - 1); k_s += (1 * WIDTH)) {
        __syncthreads();
        if ((k_s+tx) < P) {
            if (j < N) {
                C_c[tx] = C_at((k_s+tx),j);
            } // end if-else
        } // end if-else
        __syncthreads();

        for (int k = k_s; k <= ((k_s + WIDTH) - 1); k += 1) {
            if (i < M) {
                if (k < P) {
                    acc1 += (B_at(i,k) * C_c[(k - k_s)]);
                } // end if-else
            } // end if-else
        } // end for (counter k)
    } // end for (counter k_s)

    if (i < M) {
        if (j < N) {
            A_at(i,j) = acc1;
        } // end if-else
    } // end if-else
}
```

Figure 4.13: mm_cond_2: Staged & cached accumulation parallel MATMUL kernel (any matrix size)

Macros:

$X_{at}(i,j) \equiv (*(float*)(char*)X + j*pitch_X) + i$

$WIDTH \equiv 64$ $UNROLL1 \equiv 1$ $UNROLL2 \equiv 8$

```
__global__ void matmul( float* A, float* B, float* C )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;
    int tx = threadIdx.x;

    __shared__ float C_c_1[WIDTH];
    __shared__ float C_c_2[WIDTH];
    __shared__ float C_c_3[WIDTH];
    __shared__ float C_c_4[WIDTH];
    __shared__ float C_c_5[WIDTH];
    __shared__ float C_c_6[WIDTH];
    __shared__ float C_c_7[WIDTH];
    __shared__ float C_c_8[WIDTH];

    float acc1_1_1;
    float acc1_1_2;
    float acc1_1_3;
    float acc1_1_4;
    float acc1_1_5;
    float acc1_1_6;
    float acc1_1_7;
    float acc1_1_8;
    if (i < M) {
        if (j < N) {
            acc1_1_1 = A_at(i,j);
        } // end if-else
        if ((j + 1) < N) {
            acc1_1_2 = A_at(i,(j + 1));
        } // end if-else
        if ((j + 2) < N) {
            acc1_1_3 = A_at(i,(j + 2));
        } // end if-else
        if ((j + 3) < N) {
            acc1_1_4 = A_at(i,(j + 3));
        } // end if-else
        if ((j + 4) < N) {
            acc1_1_5 = A_at(i,(j + 4));
        } // end if-else
        if ((j + 5) < N) {
            acc1_1_6 = A_at(i,(j + 5));
        } // end if-else
        if ((j + 6) < N) {
            acc1_1_7 = A_at(i,(j + 6));
        } // end if-else
        if ((j + 7) < N) {
            acc1_1_8 = A_at(i,(j + 7));
        } // end if-else
    } // end if-else

    for (int k_s = 0; k_s <= (P - 1); k_s += ((1 * WIDTH) * 1)) {
        __syncthreads();
        if ((k_s+tx) < P) {
            if (j < N) {
                C_c_1[tx] = C_at((k_s+tx),j);
            } // end if-else
            if ((j + 1) < N) {
                C_c_2[tx] = C_at((k_s+tx),(j + 1));
            } // end if-else
            if ((j + 2) < N) {
                C_c_3[tx] = C_at((k_s+tx),(j + 2));
            } // end if-else
        }
    }
}
```

```

        if ((j + 3) < N) {
            C_c_4[tx] = C_at((k_s+tx), (j + 3));
        } // end if-else
        if ((j + 4) < N) {
            C_c_5[tx] = C_at((k_s+tx), (j + 4));
        } // end if-else
        if ((j + 5) < N) {
            C_c_6[tx] = C_at((k_s+tx), (j + 5));
        } // end if-else
        if ((j + 6) < N) {
            C_c_7[tx] = C_at((k_s+tx), (j + 6));
        } // end if-else
        if ((j + 7) < N) {
            C_c_8[tx] = C_at((k_s+tx), (j + 7));
        } // end if-else
    } // end if-else
    __syncthreads();

#pragma unroll 16
for (int k = k_s; k <= ((k_s + WIDTH) - 1); k += (1 * 1)) {
    if (i < M) {
        if (k < P) {
            acc1_1_1 += (B_at(i,k) * C_c_1[(k - k_s)]);
            acc1_1_2 += (B_at(i,k) * C_c_2[(k - k_s)]);
            acc1_1_3 += (B_at(i,k) * C_c_3[(k - k_s)]);
            acc1_1_4 += (B_at(i,k) * C_c_4[(k - k_s)]);
            acc1_1_5 += (B_at(i,k) * C_c_5[(k - k_s)]);
            acc1_1_6 += (B_at(i,k) * C_c_6[(k - k_s)]);
            acc1_1_7 += (B_at(i,k) * C_c_7[(k - k_s)]);
            acc1_1_8 += (B_at(i,k) * C_c_8[(k - k_s)]);
        } // end if-else
    } // end if-else
} // end for (counter k)

if (i < M) {
    if (j < N) {
        A_at(i,j) = acc1_1_1;
    } // end if-else
    if ((j + 1) < N) {
        A_at(i, (j + 1)) = acc1_1_2;
    } // end if-else
    if ((j + 2) < N) {
        A_at(i, (j + 2)) = acc1_1_3;
    } // end if-else
    if ((j + 3) < N) {
        A_at(i, (j + 3)) = acc1_1_4;
    } // end if-else
    if ((j + 4) < N) {
        A_at(i, (j + 4)) = acc1_1_5;
    } // end if-else
    if ((j + 5) < N) {
        A_at(i, (j + 5)) = acc1_1_6;
    } // end if-else
    if ((j + 6) < N) {
        A_at(i, (j + 6)) = acc1_1_7;
    } // end if-else
    if ((j + 7) < N) {
        A_at(i, (j + 7)) = acc1_1_8;
    } // end if-else
} // end if-else
}

```

Figure 4.14: mm_cond_autogpu: Optimal MATMUL kernel (any matrix size)

It can easily be understood by the kernels produced so far that not all conditional statements are optimally placed or even necessary to maintain the kernel's computational correctness. The matrix bounds-checking transformation is provided to the user solely as a facility to produce a more general kernel. To obtain the optimal general kernel, the user should either alter the resulting source code (e.g. by removing unnecessary *if* statements, or otherwise changing them), or provide the *AutoGPU* program with an input that checks for out-of-bounds references and skip the relevant transformation.

Another interesting (although expected) observation is that the kernel's optimal loop parameters are different between the `mm_autogpu` and `mm_cond_autogpu` kernels, especially the CUDA block width, which resulted in the low value of 64 threads per block. Bearing that in mind, it is possible that yet another parameter set may be decided upon, should the user supply the *AutoGPU* program with an input AST already enhanced with the necessary conditionals, as was proposed above.

Again, the execution time and performance of the discussed kernels are presented in figures 4.15 and 4.15, respectively. Logically enough, the performance of these kernels is rather lower than that of their non-bounds-checking counterparts.

Reference kernel: `mm_cond_wolfe`

The source code for the reference kernel which the optimal kernel, resulting from the *AutoGPU* program, is compared against can be seen in figure 4.17.

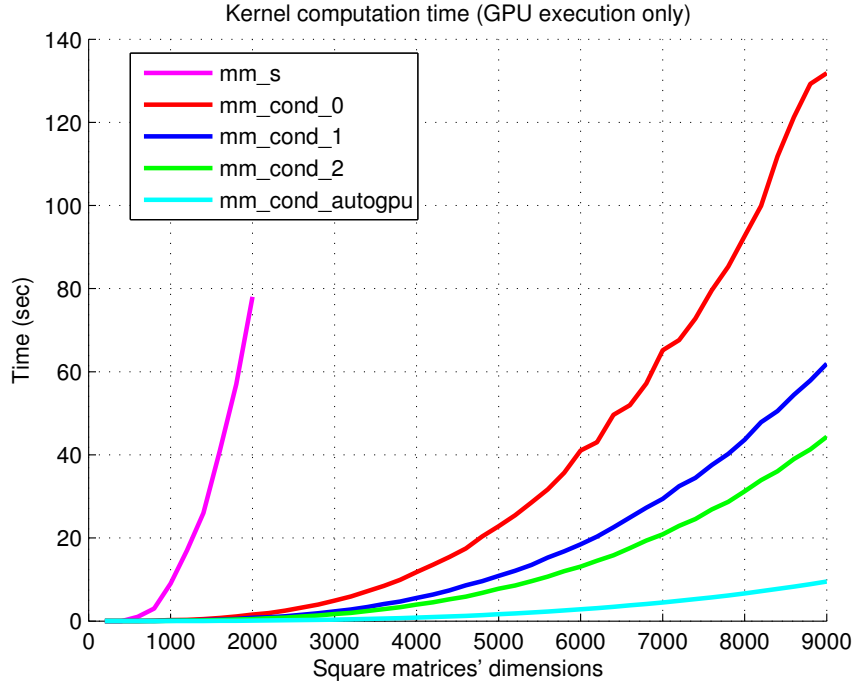


Figure 4.15: Execution times of the input, intermediate and final MATMUL kernels with conditionals

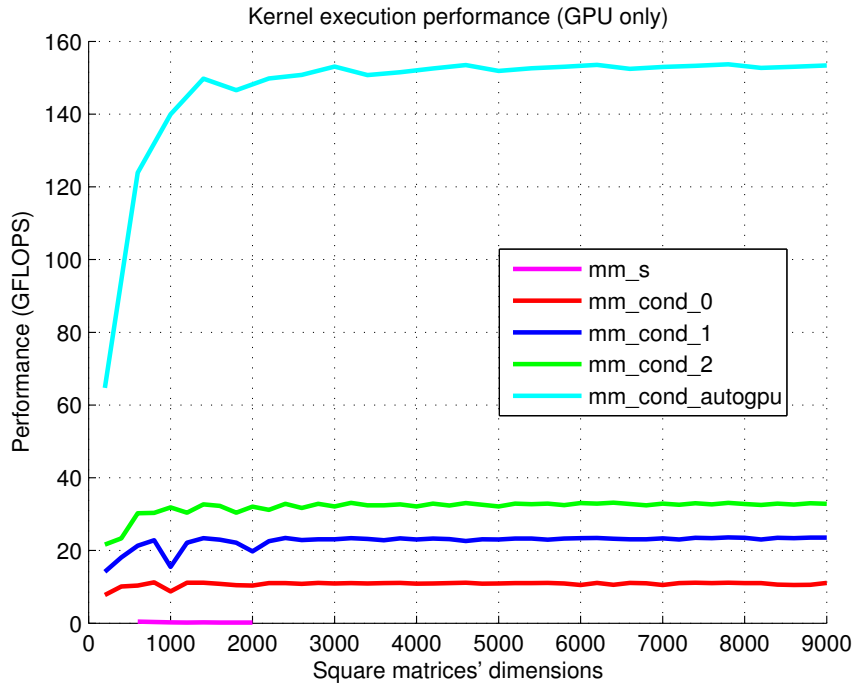


Figure 4.16: Performance of the input, intermediate and final MATMUL kernels with conditionals

```

__global__ void matmul( float* a, float* b, float* c,
                        int pitch_a, int pitch_b, int pitch_c,
                        int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*128 + tx;
    int j = blockIdx.y*4;
    __shared__ float cb0[128], cb1[128], cb2[128], cb3[128];

    float sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
    for( int ks = 0; ks < p; ks += 128 ){
        if( ks+tx < p && j < m )
            cb0[tx] = c[ks+tx+pitch_c*j];
        else
            cb0[tx] = 0.0;
        if( ks+tx < p && j+1 < m )
            cb1[tx] = c[ks+tx+pitch_c*(j+1)];
        else
            cb1[tx] = 0.0;
        if( ks+tx < p && j+2 < m )
            cb2[tx] = c[ks+tx+pitch_c*(j+2)];
        else
            cb2[tx] = 0.0;
        if( ks+tx < p && j+3 < m )
            cb3[tx] = c[ks+tx+pitch_c*(j+3)];
        else
            cb3[tx] = 0.0;
        __syncthreads();
        if( i < n ){
            int k, kk;
            for( k=ks, kk=0; k+3 < ((ks+128 < m)? ks+128 : m); k += 4, kk += 4 ){
                float rb = b[i+pitch_b*(k)];
                sum0 += rb * cb0[kk];
                sum1 += rb * cb1[kk];
                sum2 += rb * cb2[kk];
                sum3 += rb * cb3[kk];
                rb = b[i+pitch_b*(k+1)];
                sum0 += rb * cb0[kk+1];
                sum1 += rb * cb1[kk+1];
                sum2 += rb * cb2[kk+1];
                sum3 += rb * cb3[kk+1];
                rb = b[i+pitch_b*(k+2)];
                sum0 += rb * cb0[kk+2];
                sum1 += rb * cb1[kk+2];
                sum2 += rb * cb2[kk+2];
                sum3 += rb * cb3[kk+2];
                rb = b[i+pitch_b*(k+3)];
                sum0 += rb * cb0[kk+3];
                sum1 += rb * cb1[kk+3];
                sum2 += rb * cb2[kk+3];
                sum3 += rb * cb3[kk+3];
            }
        }
    }
}

```

```

    if( ks+128 > m ){
        for( ; k < m; ++k, ++kk ){
            float rb = b[i+pitch_b*(k)];
            sum0 += rb * cb0[kk];
            sum1 += rb * cb1[kk];
            sum2 += rb * cb2[kk];
            sum3 += rb * cb3[kk];
        }
    }
    __syncthreads();
}

if( i < n && j < m )
    a[i+pitch_a*j] = sum0;
if( i < n && j + 1 < m )
    a[i+pitch_a*(j+1)] = sum1;
if( i < n && j + 2 < m )
    a[i+pitch_a*(j+2)] = sum2;
if( i < n && j + 3 < m )
    a[i+pitch_a*(j+3)] = sum3;
}

```

Figure 4.17: `mm_cond_wolfe`: Michael Wolfe’s kernel (any matrix size)

The optimizations that have been applied to the optimal generated kernel and the reference one, as well as the loop parameters for each, are summarized in table 4.6.

Kernel	Block width	Staging	__shared__ caching	Loop unroll factors		
				<i>i</i> loop	<i>j</i> loop	<i>k</i> loop
<code>mm_cond_autogpu</code>	64	Yes	Yes	1	8	16
<code>mm_cond_wolfe</code>	128	Yes	Yes	1	4	4

Table 4.6: Optimal and reference **MATMUL** kernels (any matrix size) specifications

The performances of the optimal and the reference kernels are graphically presented in figures 4.18 and 4.19, for input matrices of sizes 200×200 to 16800×16800 , with a size stride of 200.

The `mm_cond_wolfe` kernel’s performance seems to be vary greatly depending on the input matrix sizes, whereas the performance of kernel `mm_cond_autogpu` is stable. This should be a result of the method used in unrolling the inner *k* loop in `mm_cond_wolfe`, as this is the major difference between the two kernels. Despite this deficiency of the `mm_cond_autogpu` kernels, however, its performance (constant, around 155 GFLOPS) is still better than the average performance achieved by the reference kernel, whose performance varies from less than 120 GFLOPS to about 170 GFLOPS.

Kernel Precision

Table 4.7 lists the maximum absolute error in the product matrix computed in the GPU (as compared to that of the CPU), for different input matrix sizes.

Conclusions

The quantitative relative performance of the kernels discussed in this subsection are presented for input matrices of size 3000×3000 , regarding execution time (table 4.8) and floating-point operations (table 4.9). The reference kernel is omitted in these tables, since no one input matrix size is representative of its performance.

As with the optimal kernel discussed in the previous subsection, a decrease of over 90% in execution time and increase of over 10x in floating-point operations is achieved by the kernel

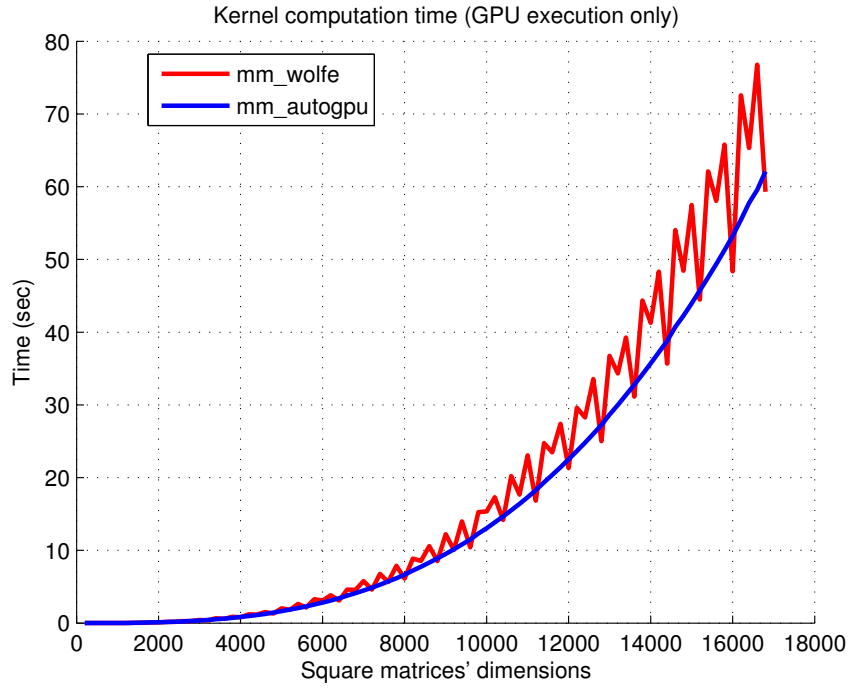


Figure 4.18: Execution times of the reference and optimal MATMUL kernels with conditionals

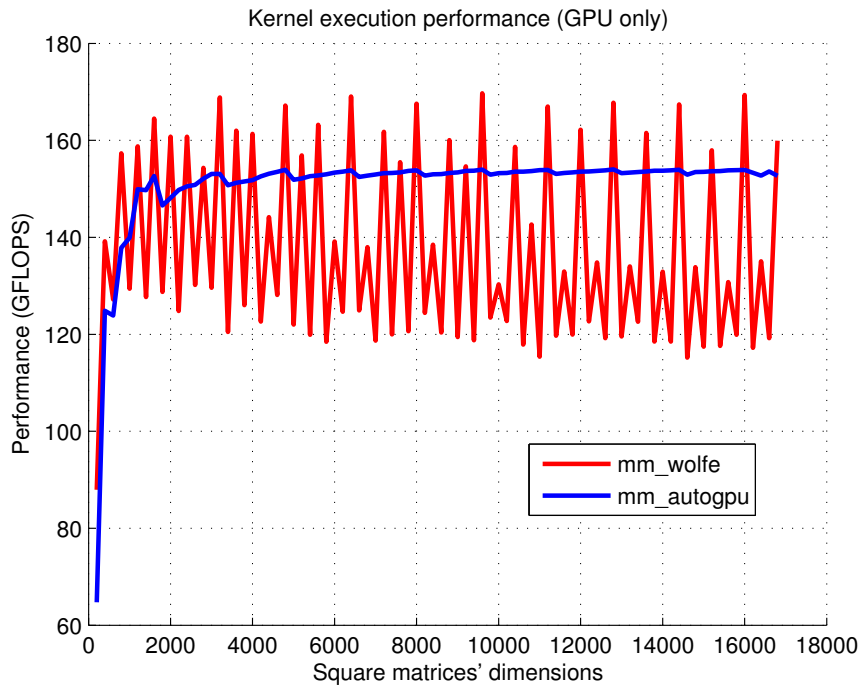


Figure 4.19: Performance of the reference and optimal MATMUL kernels with conditionals

Matrix size	Maximum absolute error
400×400	$9.53674 \cdot 10^{-6}$
800×800	$2.09808 \cdot 10^{-5}$
1600×1600	$3.05176 \cdot 10^{-5}$
3000×3000	$5.34058 \cdot 10^{-5}$

Table 4.7: MATMUL kernel (with conditionals) output’s maximum absolute error

Kernel	Execution time	Approximate speed-up		
		to mm_cond_0	to mm_cond_1	to mm_cond_2
mm_cond_0	4925 msec	–	–	–
mm_cond_1	2338 msec	53%	–	–
mm_cond_2	1681 msec	66%	28%	–
mm_cond_autogpu	389 msec	92%	84%	77%

Table 4.8: MATMUL kernel execution times and relative speed-up for 3000×3000 matrices

Kernel	Performance	Approximate speed-up		
		to mm_cond_0	to mm_cond_1	to mm_cond_2
mm_cond_0	11 GFLOPS	–	–	–
mm_cond_1	23 GFLOPS	109%	–	–
mm_cond_2	32 GFLOPS	191%	39%	–
mm_cond_autogpu	139 GFLOPS	1164%	504%	334%

Table 4.9: MATMUL kernel FLOPS and relative speed-up for 3000×3000 matrices

generated by the *AutoGPU* program. Also, the resulting statistics suggest that the presence of the conditional statements limits the gain achieved by caching the **C** matrix references through the `__shared__` memory space.

In conclusion, the simple conditional statement generation method employed by the *AutoGPU* program does not automatically result to an optimal kernel; notwithstanding, the optimized one that is produced still demonstrates a high performance.

4.2 Direct 2D Convolution: CONV

In this section, the process of optimizing a kernel that computes convolution of a 2D image with a template will be presented, as it is performed by the *AutoGPU* program.

The direct convolution algorithm is another example of a computation-intensive procedure that be mapped to the CUDA programming model. As with the MATMUL kernel (section 4.1), each thread will be calculating the value of 1 element of the resulting matrix, multiplying its neighboring elements with corresponding elements of the convolution template.

Calculating a convolution directly is only advantageous for small template matrices, of size up to 11×11 ; beyond that, it is faster to transform the matrices in the Fourier domain (using a *Fast Fourier Transform* method) and calculate is the transformes matrices’ product. Hence, only small template matrices will be discussed in this section.

Due to the small size of the template matrix, as well as the fact that the template element access pattern is the same for each computed element of the convolution matrix, the template matrix is set to reside in `__constant__` memory: This way, since all threads will be accessing the same template value at the same time, this will result in single memory transfer, as fast as a register access. Since the template matrix will not exceed a certain size, it is safe to declare the `__constant__` template matrix with a size of, say, 225 (i.e. $15 \cdot 15$).

The computed convolution will be that of the expression $\mathbf{O} = \mathbf{I} * \mathbf{T}$. Matrices are stored in

column-major fashion and their dimension variables are as follows:

$$\mathbf{I} : IH \times IW \quad , \quad \mathbf{T} : TH \times TW \quad , \quad \mathbf{O} : SH \times SW = (IH + TH - 1) \times (IW + TW - 1)$$

In order to calculate the convolution without any conditional statements to ensure that matrix references stay within bounds (even for the serial version), the input image can be stored in an $(SH+TH-1) \times (IW+TW-1) = BH \times BW$ matrix, with an offset of $(TH-1, TW-1)$ (assuming 0-indexed arrays). This means that the image matrix will be zero-padded by $TH-1$ elements along the start of the vertical axis, and by $TW-1$ elements along the start of the horizontal axis. If that condition is satisfied, a simple serial implementation of 2D convolution is that shown in figure 4.20.

```
void convolution_serial( float* I, float* T, float* O,
                      int BH, int BW, int TH, int TW, int SH, int SW )
{
    for ( int i = 0; i < SH; ++i ) {
        for ( int j = 0; j < SW; ++j ) {

            for ( int ti = 0; ti < TH; ++ti ) {
                for ( int tj = 0; tj < TW; ++tj ) {
                    O[i + j*SH] += I[(i+ti)+(j+tj)*BH] * T[(TH-1-ti)+(TW-1-tj)*TH];
                } // end for (counter tj)
            } // end for (counter ti)

        } // end for (counter j)
    } // end for (counter i)
} // end function 'convolution_serial'
```

Figure 4.20: conv_s: Serial CONV kernel

The number of floating-point operations per convolution element calculation is not very big (specifically, it is $2 \cdot TH \cdot TW$), as is the width of the CUDA blocks worked out by the *AutoGPU*—as will be shown in the following subsections—, therefore zero-padding the matrices and allowing a (small) number of redundant threads to execute will be preferred to issuing conditional statements to control matrix access.

The appropriate zero-padding for a CUDA block which calculates the values of an $A \times B$ segment of the output matrix is as follows: First, the dimensions of the output matrix have to be rounded up to the nearest multiple of the corresponding block dimension¹. Then, the *extended* dimensions of the input image, $BH \times BW$ are calculated as before, only with the *new* values of the SH, SW variables, and the image matrix is again copied to GPU memory with an offset of $(TH-1, TW-1)$. As a result, the convolution can be computed without employing any conditional statements, for input of any size.

Since the set of valid unroll factors for the two inner *for* loops is dependent on the size of the template matrix (as these should be divisors of the template matrix dimensions), two cases of the direct 2D convolution will be studied: One where the convolution template is a 5×5 matrix, and one where it is a 11×11 matrix.

4.2.1 5×5 convolution template

In this subsection, four parallel implementations of matrix multiplication will be compared:

¹Here, “block dimension” does not refer to the dimension of the actual CUDA block configuration, but rather to the dimension of the block times the corresponding parallel loop unroll factor, since an unrolled block will compute more output elements than it has threads.

- conv5.0 : The “vanilla” parallel kernel—a direct translation of the input AST to CUDA source code.
- conv5.1 : The parallel kernel, after staging of partial accumulations.
- conv5_autogpu : The optimized parallel kernel, after unrolling of both parallel and serial loops.
- conv5_flcc : The direct convolution implementation performed by the FLCC library, developed by Georgios Papamakarios and Georgios Rizos.

Successive versions of the kernel

First, the AST is traversed in order to mark all 2D-pitched array references. The template matrix will not be so marked, since it has been set to reside in `__constant__` memory, along with all matrix dimension variables. Thus, the source code for the vanilla parallel version of the CONV kernel is obtained, shown in figure 4.21.

Macros:

`X_at(i,j) ≡ (*(float*)(char*)X + j*pitch_X) + i)`

`WIDTH ≡ 128 UNROLL1 ≡ 1 UNROLL2 ≡ 1`

```
__global__ void convolution( float* O, float* I )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;

    for (int ti = 0; ti <= (T_H - 1); ti += 1) {
        for (int tj = 0; tj <= (T_W - 1); tj += 1) {
            O_at(i,j) += (I_at((i + ti), (j + tj))
                        * T[(((T_H - 1) - ti) + (((T_W - 1) - tj) * T_H))]);
        } // end for (counter tj)
    } // end for (counter ti)
}
```

Figure 4.21: conv5.0: Vanilla parallel CONV kernel (5×5 template)

The accumulation of the convolution result straight in the `O` matrix, which resides in `__global__` memory, can be staged through an automatic variable. This results in the kernel shown in figure 4.22.

Macros:

`X_at(i,j) ≡ (*(float*)(char*)X + j*pitch_X) + i)`

`WIDTH ≡ 128 UNROLL1 ≡ 1 UNROLL2 ≡ 1`

```
__global__ void convolution( float* O, float* I )
{
    int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    int j = blockIdx.y*UNROLL2;

    float acc1;
    acc1 = O_at(i,j);

    for (int ti = 0; ti <= (T_H - 1); ti += 1) {
        for (int tj = 0; tj <= (T_W - 1); tj += 1) {
            acc1 += (I_at((i + ti), (j + tj))
                    * T[(((T_H - 1) - ti) + (((T_W - 1) - tj) * T_H))]);
        } // end for (counter tj)
    } // end for (counter ti)

    O_at(i,j) = acc1;
}
```

Figure 4.22: conv5.1: Staged accumulation parallel CONV kernel (5×5 template)

Application of the array caching transformation has no effect, since the `I` matrix reference depends on both the i and j loops, therefore the CUDA *coalescing* condition is met for that access:

Consecutive threads access consecutive array elements, resulting in a single memory transfer per *warp size* threads.

Lastly, the loops of the kernel are unrolled. With regard to the *ti* and *tj* loops, the maximal valid set of unroll factors is $\{1, 5\}$. The unroll factors' set considered for the parallel (*i* and *j* loops) is $\{1, 2, 4, 8, 16\}$. The code finally generated by the *AutoGPU* program is shown in figure 4.23.

Here, the unrolling of the serial *for* loops is shown without use of the `#pragma unroll` directive. Furthermore, to demonstrate a case of minor optimizations that may be applied after the kernel's generation, the accumulation variables are explicitly set to reside in registers and they are initialized with a floating-point literal, instead of accessing matrix *I*; also, the *i* and *j* loop counters are set as `const` values.

Macros:

$X_at(i,j) \equiv (*(float*)(char*)X + j*pitch_X) + i$)

$WIDTH \equiv 32$ $UNROLL1 \equiv 1$ $UNROLL2 \equiv 8$

```
__global__ void convolution( float* O, float* I )
{
    const int i = blockIdx.x * blockDim.x*UNROLL1 + threadIdx.x;
    const int j = blockIdx.y*UNROLL2;
    register float acc1_1_1 = 0.0f;
    register float acc1_1_2 = 0.0f;
    register float acc1_1_3 = 0.0f;
    register float acc1_1_4 = 0.0f;
    register float acc1_1_5 = 0.0f;
    register float acc1_1_6 = 0.0f;
    register float acc1_1_7 = 0.0f;
    register float acc1_1_8 = 0.0f;

    acc1_1_1 += (I_at((i),(j))
        * T[(((T_H - 1)) + (((T_W - 1)) * T_H))]);
    acc1_1_1 += (I_at((i),(j + 1))
        * T[(((T_H - 1)) + (((T_W - 1) - 1) * T_H))]);
    acc1_1_1 += (I_at((i),(j + 2))
        * T[(((T_H - 1)) + (((T_W - 1) - 2) * T_H))]);
    acc1_1_1 += (I_at((i),(j + 3))
        * T[(((T_H - 1)) + (((T_W - 1) - 3) * T_H))]);
    acc1_1_1 += (I_at((i),(j + 4))
        * T[(((T_H - 1)) + (((T_W - 1) - 4) * T_H))]);
    acc1_1_1 += (I_at((i + 1),(j))
        * T[(((T_H - 1) - 1) + (((T_W - 1)) * T_H))]);
    acc1_1_1 += (I_at((i + 1),(j + 1))
        * T[(((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H))]);
    acc1_1_1 += (I_at((i + 1),(j + 2))
        * T[(((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H))]);
    acc1_1_1 += (I_at((i + 1),(j + 3))
        * T[(((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H))]);
    acc1_1_1 += (I_at((i + 1),(j + 4))
        * T[(((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H))]);
    acc1_1_1 += (I_at((i + 2),(j))
        * T[(((T_H - 1) - 2) + (((T_W - 1)) * T_H))]);
    acc1_1_1 += (I_at((i + 2),(j + 1))
        * T[(((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H))]);
    acc1_1_1 += (I_at((i + 2),(j + 2))
        * T[(((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H))]);
    acc1_1_1 += (I_at((i + 2),(j + 3))
        * T[(((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H))]);
    acc1_1_1 += (I_at((i + 2),(j + 4))
        * T[(((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H))]);
    acc1_1_1 += (I_at((i + 3),(j))
        * T[(((T_H - 1) - 3) + (((T_W - 1)) * T_H))]);
    acc1_1_1 += (I_at((i + 3),(j + 1))
        * T[(((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H))]);
    acc1_1_1 += (I_at((i + 3),(j + 2))
        * T[(((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H))]);
    acc1_1_1 += (I_at((i + 3),(j + 3))
        * T[(((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H))]);
    acc1_1_1 += (I_at((i + 3),(j + 4))
        * T[(((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H))]);
    acc1_1_1 += (I_at((i + 4),(j))
        * T[(((T_H - 1) - 4) + (((T_W - 1)) * T_H))]);
    acc1_1_1 += (I_at((i + 4),(j + 1))
        * T[(((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H))]);
    acc1_1_1 += (I_at((i + 4),(j + 2))
        * T[(((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H))]);
    acc1_1_1 += (I_at((i + 4),(j + 3))
        * T[(((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H))]);
    acc1_1_1 += (I_at((i + 4),(j + 4))
        * T[(((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H))]);
}
```



```

acc1_1_3 += (I_at((i + 1), ((j + 2) + 1))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H))]);
acc1_1_3 += (I_at((i + 1), ((j + 2) + 2))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H))]);
acc1_1_3 += (I_at((i + 1), ((j + 2) + 3))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H))]);
acc1_1_3 += (I_at((i + 1), ((j + 2) + 4))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H))]);
acc1_1_3 += (I_at((i + 2), ((j + 2)))
* T[(((T_H - 1) - 2) + (((T_W - 1)) * T_H))]);
acc1_1_3 += (I_at((i + 2), ((j + 2) + 1))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H))]);
acc1_1_3 += (I_at((i + 2), ((j + 2) + 2))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H))]);
acc1_1_3 += (I_at((i + 2), ((j + 2) + 3))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H))]);
acc1_1_3 += (I_at((i + 2), ((j + 2) + 4))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H))]);
acc1_1_3 += (I_at((i + 3), ((j + 2)))
* T[(((T_H - 1) - 3) + (((T_W - 1)) * T_H))]);
acc1_1_3 += (I_at((i + 3), ((j + 2) + 1))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H))]);
acc1_1_3 += (I_at((i + 3), ((j + 2) + 2))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H))]);
acc1_1_3 += (I_at((i + 3), ((j + 2) + 3))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H))]);
acc1_1_3 += (I_at((i + 3), ((j + 2) + 4))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H))]);
acc1_1_3 += (I_at((i + 4), ((j + 2)))
* T[(((T_H - 1) - 4) + (((T_W - 1)) * T_H))]);
acc1_1_3 += (I_at((i + 4), ((j + 2) + 1))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H))]);
acc1_1_3 += (I_at((i + 4), ((j + 2) + 2))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H))]);
acc1_1_3 += (I_at((i + 4), ((j + 2) + 3))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H))]);
acc1_1_3 += (I_at((i + 4), ((j + 2) + 4))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H))]);
acc1_1_4 += (I_at((i), ((j + 3)))
* T[(((T_H - 1)) + (((T_W - 1)) * T_H))]);
acc1_1_4 += (I_at((i), ((j + 3) + 1))
* T[(((T_H - 1)) + (((T_W - 1) - 1) * T_H))]);
acc1_1_4 += (I_at((i), ((j + 3) + 2))
* T[(((T_H - 1)) + (((T_W - 1) - 2) * T_H))]);
acc1_1_4 += (I_at((i), ((j + 3) + 3))
* T[(((T_H - 1)) + (((T_W - 1) - 3) * T_H))]);
acc1_1_4 += (I_at((i), ((j + 3) + 4))
* T[(((T_H - 1)) + (((T_W - 1) - 4) * T_H))]);
acc1_1_4 += (I_at((i + 1), ((j + 3)))
* T[(((T_H - 1) - 1) + (((T_W - 1)) * T_H))]);
acc1_1_4 += (I_at((i + 1), ((j + 3) + 1))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H))]);
acc1_1_4 += (I_at((i + 1), ((j + 3) + 2))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H))]);
acc1_1_4 += (I_at((i + 1), ((j + 3) + 3))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H))]);
acc1_1_4 += (I_at((i + 1), ((j + 3) + 4))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H))]);
acc1_1_4 += (I_at((i + 2), ((j + 3)))
* T[(((T_H - 1) - 2) + (((T_W - 1)) * T_H))]);

```

```

acc1_1_4 += (I_at((i + 2), ((j + 3) + 1))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H))]);
acc1_1_4 += (I_at((i + 2), ((j + 3) + 2))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H))]);
acc1_1_4 += (I_at((i + 2), ((j + 3) + 3))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H))]);
acc1_1_4 += (I_at((i + 2), ((j + 3) + 4))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H))]);
acc1_1_4 += (I_at((i + 3), ((j + 3)))
            * T[(((T_H - 1) - 3) + (((T_W - 1)) * T_H))]);
acc1_1_4 += (I_at((i + 3), ((j + 3) + 1))
            * T[(((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H))]);
acc1_1_4 += (I_at((i + 3), ((j + 3) + 2))
            * T[(((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H))]);
acc1_1_4 += (I_at((i + 3), ((j + 3) + 3))
            * T[(((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H))]);
acc1_1_4 += (I_at((i + 3), ((j + 3) + 4))
            * T[(((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H))]);
acc1_1_4 += (I_at((i + 4), ((j + 3)))
            * T[(((T_H - 1) - 4) + (((T_W - 1)) * T_H))]);
acc1_1_4 += (I_at((i + 4), ((j + 3) + 1))
            * T[(((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H))]);
acc1_1_4 += (I_at((i + 4), ((j + 3) + 2))
            * T[(((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H))]);
acc1_1_4 += (I_at((i + 4), ((j + 3) + 3))
            * T[(((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H))]);
acc1_1_4 += (I_at((i + 4), ((j + 3) + 4))
            * T[(((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H))]);
acc1_1_5 += (I_at((i), ((j + 4)))
            * T[(((T_H - 1)) + (((T_W - 1)) * T_H))]);
acc1_1_5 += (I_at((i), ((j + 4) + 1))
            * T[(((T_H - 1)) + (((T_W - 1) - 1) * T_H))]);
acc1_1_5 += (I_at((i), ((j + 4) + 2))
            * T[(((T_H - 1)) + (((T_W - 1) - 2) * T_H))]);
acc1_1_5 += (I_at((i), ((j + 4) + 3))
            * T[(((T_H - 1)) + (((T_W - 1) - 3) * T_H))]);
acc1_1_5 += (I_at((i), ((j + 4) + 4))
            * T[(((T_H - 1)) + (((T_W - 1) - 4) * T_H))]);
acc1_1_5 += (I_at((i + 1), ((j + 4)))
            * T[(((T_H - 1) - 1) + (((T_W - 1)) * T_H))]);
acc1_1_5 += (I_at((i + 1), ((j + 4) + 1))
            * T[(((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H))]);
acc1_1_5 += (I_at((i + 1), ((j + 4) + 2))
            * T[(((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H))]);
acc1_1_5 += (I_at((i + 1), ((j + 4) + 3))
            * T[(((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H))]);
acc1_1_5 += (I_at((i + 1), ((j + 4) + 4))
            * T[(((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H))]);
acc1_1_5 += (I_at((i + 2), ((j + 4)))
            * T[(((T_H - 1) - 2) + (((T_W - 1)) * T_H))]);
acc1_1_5 += (I_at((i + 2), ((j + 4) + 1))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H))]);
acc1_1_5 += (I_at((i + 2), ((j + 4) + 2))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H))]);
acc1_1_5 += (I_at((i + 2), ((j + 4) + 3))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H))]);
acc1_1_5 += (I_at((i + 2), ((j + 4) + 4))
            * T[(((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H))]);
acc1_1_5 += (I_at((i + 3), ((j + 4)))
            * T[(((T_H - 1) - 3) + (((T_W - 1)) * T_H))]);

```

```

acc1_1_5 += (I_at((i + 3), ((j + 4) + 1))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H))]);
acc1_1_5 += (I_at((i + 3), ((j + 4) + 2))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H))]);
acc1_1_5 += (I_at((i + 3), ((j + 4) + 3))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H))]);
acc1_1_5 += (I_at((i + 3), ((j + 4) + 4))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H))]);
acc1_1_5 += (I_at((i + 4), ((j + 4)))
* T[(((T_H - 1) - 4) + (((T_W - 1)) * T_H))]);
acc1_1_5 += (I_at((i + 4), ((j + 4) + 1))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H))]);
acc1_1_5 += (I_at((i + 4), ((j + 4) + 2))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H))]);
acc1_1_5 += (I_at((i + 4), ((j + 4) + 3))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H))]);
acc1_1_5 += (I_at((i + 4), ((j + 4) + 4))
* T[(((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H))]);
acc1_1_6 += (I_at((i), ((j + 5)))
* T[(((T_H - 1)) + (((T_W - 1)) * T_H))]);
acc1_1_6 += (I_at((i), ((j + 5) + 1))
* T[(((T_H - 1)) + (((T_W - 1) - 1) * T_H))]);
acc1_1_6 += (I_at((i), ((j + 5) + 2))
* T[(((T_H - 1)) + (((T_W - 1) - 2) * T_H))]);
acc1_1_6 += (I_at((i), ((j + 5) + 3))
* T[(((T_H - 1)) + (((T_W - 1) - 3) * T_H))]);
acc1_1_6 += (I_at((i), ((j + 5) + 4))
* T[(((T_H - 1)) + (((T_W - 1) - 4) * T_H))]);
acc1_1_6 += (I_at((i + 1), ((j + 5)))
* T[(((T_H - 1) - 1) + (((T_W - 1)) * T_H))]);
acc1_1_6 += (I_at((i + 1), ((j + 5) + 1))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H))]);
acc1_1_6 += (I_at((i + 1), ((j + 5) + 2))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H))]);
acc1_1_6 += (I_at((i + 1), ((j + 5) + 3))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H))]);
acc1_1_6 += (I_at((i + 1), ((j + 5) + 4))
* T[(((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H))]);
acc1_1_6 += (I_at((i + 2), ((j + 5)))
* T[(((T_H - 1) - 2) + (((T_W - 1)) * T_H))]);
acc1_1_6 += (I_at((i + 2), ((j + 5) + 1))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H))]);
acc1_1_6 += (I_at((i + 2), ((j + 5) + 2))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H))]);
acc1_1_6 += (I_at((i + 2), ((j + 5) + 3))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H))]);
acc1_1_6 += (I_at((i + 2), ((j + 5) + 4))
* T[(((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H))]);
acc1_1_6 += (I_at((i + 3), ((j + 5)))
* T[(((T_H - 1) - 3) + (((T_W - 1)) * T_H))]);
acc1_1_6 += (I_at((i + 3), ((j + 5) + 1))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H))]);
acc1_1_6 += (I_at((i + 3), ((j + 5) + 2))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H))]);
acc1_1_6 += (I_at((i + 3), ((j + 5) + 3))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H))]);
acc1_1_6 += (I_at((i + 3), ((j + 5) + 4))
* T[(((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H))]);
acc1_1_6 += (I_at((i + 4), ((j + 5)))
* T[(((T_H - 1) - 4) + (((T_W - 1)) * T_H))]);

```

```

acc1_1_6 += (I_at((i + 4), ((j + 5) + 1))
* T[((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H)]);
acc1_1_6 += (I_at((i + 4), ((j + 5) + 2))
* T[((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H)]);
acc1_1_6 += (I_at((i + 4), ((j + 5) + 3))
* T[((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H)]);
acc1_1_6 += (I_at((i + 4), ((j + 5) + 4))
* T[((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H)]);
acc1_1_7 += (I_at((i), ((j + 6)))
* T[((T_H - 1)) + (((T_W - 1)) * T_H)]);
acc1_1_7 += (I_at((i), ((j + 6) + 1))
* T[((T_H - 1)) + (((T_W - 1) - 1) * T_H)]);
acc1_1_7 += (I_at((i), ((j + 6) + 2))
* T[((T_H - 1)) + (((T_W - 1) - 2) * T_H)]);
acc1_1_7 += (I_at((i), ((j + 6) + 3))
* T[((T_H - 1)) + (((T_W - 1) - 3) * T_H)]);
acc1_1_7 += (I_at((i), ((j + 6) + 4))
* T[((T_H - 1)) + (((T_W - 1) - 4) * T_H)]);
acc1_1_7 += (I_at((i + 1), ((j + 6)))
* T[((T_H - 1) - 1) + (((T_W - 1)) * T_H)]);
acc1_1_7 += (I_at((i + 1), ((j + 6) + 1))
* T[((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H)]);
acc1_1_7 += (I_at((i + 1), ((j + 6) + 2))
* T[((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H)]);
acc1_1_7 += (I_at((i + 1), ((j + 6) + 3))
* T[((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H)]);
acc1_1_7 += (I_at((i + 1), ((j + 6) + 4))
* T[((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H)]);
acc1_1_7 += (I_at((i + 2), ((j + 6)))
* T[((T_H - 1) - 2) + (((T_W - 1)) * T_H)]);
acc1_1_7 += (I_at((i + 2), ((j + 6) + 1))
* T[((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H)]);
acc1_1_7 += (I_at((i + 2), ((j + 6) + 2))
* T[((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H)]);
acc1_1_7 += (I_at((i + 2), ((j + 6) + 3))
* T[((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H)]);
acc1_1_7 += (I_at((i + 2), ((j + 6) + 4))
* T[((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H)]);
acc1_1_7 += (I_at((i + 3), ((j + 6)))
* T[((T_H - 1) - 3) + (((T_W - 1)) * T_H)]);
acc1_1_7 += (I_at((i + 3), ((j + 6) + 1))
* T[((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H)]);
acc1_1_7 += (I_at((i + 3), ((j + 6) + 2))
* T[((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H)]);
acc1_1_7 += (I_at((i + 3), ((j + 6) + 3))
* T[((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H)]);
acc1_1_7 += (I_at((i + 3), ((j + 6) + 4))
* T[((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H)]);
acc1_1_7 += (I_at((i + 4), ((j + 6)))
* T[((T_H - 1) - 4) + (((T_W - 1)) * T_H)]);
acc1_1_7 += (I_at((i + 4), ((j + 6) + 1))
* T[((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H)]);
acc1_1_7 += (I_at((i + 4), ((j + 6) + 2))
* T[((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H)]);
acc1_1_7 += (I_at((i + 4), ((j + 6) + 3))
* T[((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H)]);
acc1_1_7 += (I_at((i + 4), ((j + 6) + 4))
* T[((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H)]);
acc1_1_8 += (I_at((i), ((j + 7)))
* T[((T_H - 1)) + (((T_W - 1)) * T_H)]);

```

```

acc1_1_8 += (I_at((i), ((j + 7) + 1))
            * T[((T_H - 1)) + (((T_W - 1) - 1) * T_H)]);
acc1_1_8 += (I_at((i), ((j + 7) + 2))
            * T[((T_H - 1)) + (((T_W - 1) - 2) * T_H)]);
acc1_1_8 += (I_at((i), ((j + 7) + 3))
            * T[((T_H - 1)) + (((T_W - 1) - 3) * T_H)]);
acc1_1_8 += (I_at((i), ((j + 7) + 4))
            * T[((T_H - 1)) + (((T_W - 1) - 4) * T_H)]);
acc1_1_8 += (I_at((i + 1), ((j + 7)))
            * T[((T_H - 1) - 1) + (((T_W - 1)) * T_H)]);
acc1_1_8 += (I_at((i + 1), ((j + 7) + 1))
            * T[((T_H - 1) - 1) + (((T_W - 1) - 1) * T_H)]);
acc1_1_8 += (I_at((i + 1), ((j + 7) + 2))
            * T[((T_H - 1) - 1) + (((T_W - 1) - 2) * T_H)]);
acc1_1_8 += (I_at((i + 1), ((j + 7) + 3))
            * T[((T_H - 1) - 1) + (((T_W - 1) - 3) * T_H)]);
acc1_1_8 += (I_at((i + 1), ((j + 7) + 4))
            * T[((T_H - 1) - 1) + (((T_W - 1) - 4) * T_H)]);
acc1_1_8 += (I_at((i + 2), ((j + 7)))
            * T[((T_H - 1) - 2) + (((T_W - 1)) * T_H)]);
acc1_1_8 += (I_at((i + 2), ((j + 7) + 1))
            * T[((T_H - 1) - 2) + (((T_W - 1) - 1) * T_H)]);
acc1_1_8 += (I_at((i + 2), ((j + 7) + 2))
            * T[((T_H - 1) - 2) + (((T_W - 1) - 2) * T_H)]);
acc1_1_8 += (I_at((i + 2), ((j + 7) + 3))
            * T[((T_H - 1) - 2) + (((T_W - 1) - 3) * T_H)]);
acc1_1_8 += (I_at((i + 2), ((j + 7) + 4))
            * T[((T_H - 1) - 2) + (((T_W - 1) - 4) * T_H)]);
acc1_1_8 += (I_at((i + 3), ((j + 7)))
            * T[((T_H - 1) - 3) + (((T_W - 1)) * T_H)]);
acc1_1_8 += (I_at((i + 3), ((j + 7) + 1))
            * T[((T_H - 1) - 3) + (((T_W - 1) - 1) * T_H)]);
acc1_1_8 += (I_at((i + 3), ((j + 7) + 2))
            * T[((T_H - 1) - 3) + (((T_W - 1) - 2) * T_H)]);
acc1_1_8 += (I_at((i + 3), ((j + 7) + 3))
            * T[((T_H - 1) - 3) + (((T_W - 1) - 3) * T_H)]);
acc1_1_8 += (I_at((i + 3), ((j + 7) + 4))
            * T[((T_H - 1) - 3) + (((T_W - 1) - 4) * T_H)]);
acc1_1_8 += (I_at((i + 4), ((j + 7)))
            * T[((T_H - 1) - 4) + (((T_W - 1)) * T_H)]);
acc1_1_8 += (I_at((i + 4), ((j + 7) + 1))
            * T[((T_H - 1) - 4) + (((T_W - 1) - 1) * T_H)]);
acc1_1_8 += (I_at((i + 4), ((j + 7) + 2))
            * T[((T_H - 1) - 4) + (((T_W - 1) - 2) * T_H)]);
acc1_1_8 += (I_at((i + 4), ((j + 7) + 3))
            * T[((T_H - 1) - 4) + (((T_W - 1) - 3) * T_H)]);
acc1_1_8 += (I_at((i + 4), ((j + 7) + 4))
            * T[((T_H - 1) - 4) + (((T_W - 1) - 4) * T_H)]);

O_at(i, j) = acc1_1_1;
O_at(i, (j + 1)) = acc1_1_2;
O_at(i, (j + 2)) = acc1_1_3;
O_at(i, (j + 3)) = acc1_1_4;
O_at(i, (j + 4)) = acc1_1_5;
O_at(i, (j + 5)) = acc1_1_6;
O_at(i, (j + 6)) = acc1_1_7;
O_at(i, (j + 7)) = acc1_1_8;
}

```

Figure 4.23: conv5_autogpu: Optimal CONV kernel (5×5 template)

Figures 4.24 and 4.25 show the execution time and FLOPS achieved by the serial and each parallel implementation of direct 2D convolution algorithm, for square input images of increasing

size.

Reference kernel: `conv5_flcc`

The FLCC library defines (among others) a set of direct 2D convolution kernels for square templates of sizes 1×1 up to 32×32 . According to the FLCC model, the template matrix is not stored in `__constant__` memory, but rather in `__global__` memory and is transferred in the `__shared__` memory space at the kernel's onset.

This is not the only difference between the optimal generated kernel and the FLCC implementation: The FLCC library kernel operates on 2-dimensional CUDA blocks, caches the image matrix through `__shared__` memory, and the parallel loops are not unrolled. The source code of the FLCC implementation of the direct 2D convolution is shown in figure 4.26.

The optimizations that have been applied to the optimal generated kernel and the reference one, as well as the loop parameters for each, are summarized in table 4.10.

Kernel	Block size	Staging	<code>__shared__</code> caching	Loop unroll factors			
				<i>i</i> loop	<i>j</i> loop	<i>ti</i> loop	<i>tj</i> loop
<code>conv5_autogpu</code>	32×1	Yes	No	1	8	5	5
<code>conv5_flcc</code>	16×16	Yes	Yes	1	1	5	5

Table 4.10: Optimal and reference CONV kernels specifications (5×5 template)

Graphical performance statistics for each of the two kernels are presented in figures 4.27 and 4.28. These were obtained for input image matrices of sizes 128×128 to 17280×17280 .

Kernel precision

Table 4.11 lists the maximum absolute error in the convolution matrix computed in the GPU (as compared to that of the CPU), for different input sizes sizes.

Image size	Maximum absolute error
1024×1024	$1.90735 \cdot 10^{-6}$
2048×2048	$1.90735 \cdot 10^{-6}$
4096×4096	$2.38419 \cdot 10^{-6}$
8192×8192	$2.38419 \cdot 10^{-6}$
16384×16384	$2.38419 \cdot 10^{-6}$

Table 4.11: CONV kernel output's maximum absolute error (5×5 template)

Conclusions

A quantitative comparison of the kernels discussed in this subsection, for an input image matrix of size 8192×8192 , is presented in tables 4.12 and 4.13, regarding execution time and floating-point operations, respectively.

It can be seen that the various optimizations –even though these are different for the `conv5_flcc` `conv5_autogpu` kernels– again result in a considerable performance boost, up to almost 5x in the GFLOPS performed.

Interestingly enough, the optimal generated kernel's usage of the GPU multiprocessores seems to be getting better as the input image size increases, while the opposite is observed for the reference kernel, as figure 4.28 suggests.

4.2.2 11×11 convolution template

In this subsection, the direct 2D convolution kernel is again optimized, but with an 11×11 template matrix as input. This difference in the template size affects the values that the unroll factors of

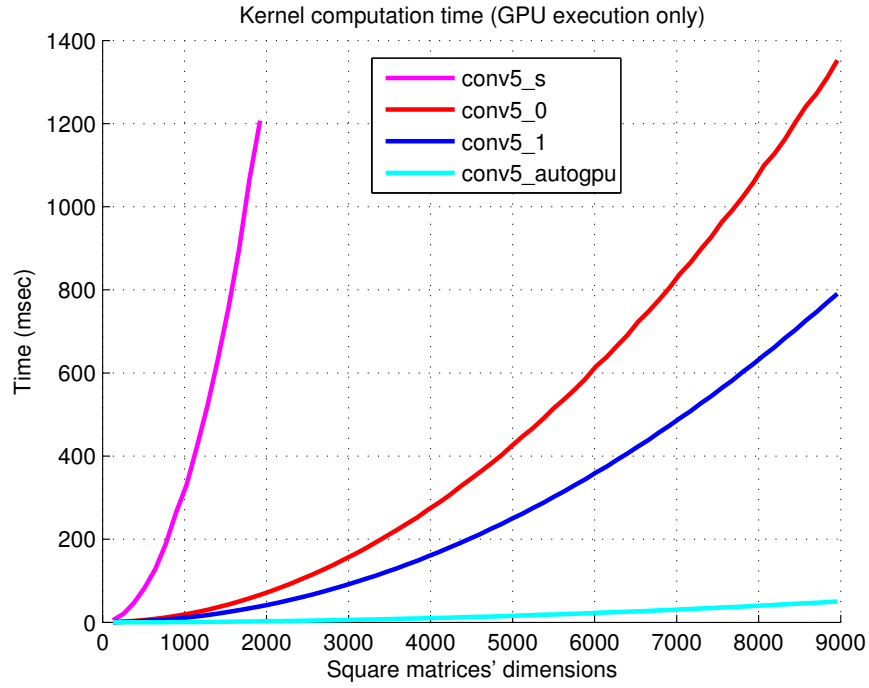


Figure 4.24: Execution times of the input, intermediate and final CONV kernels (5×5 template)

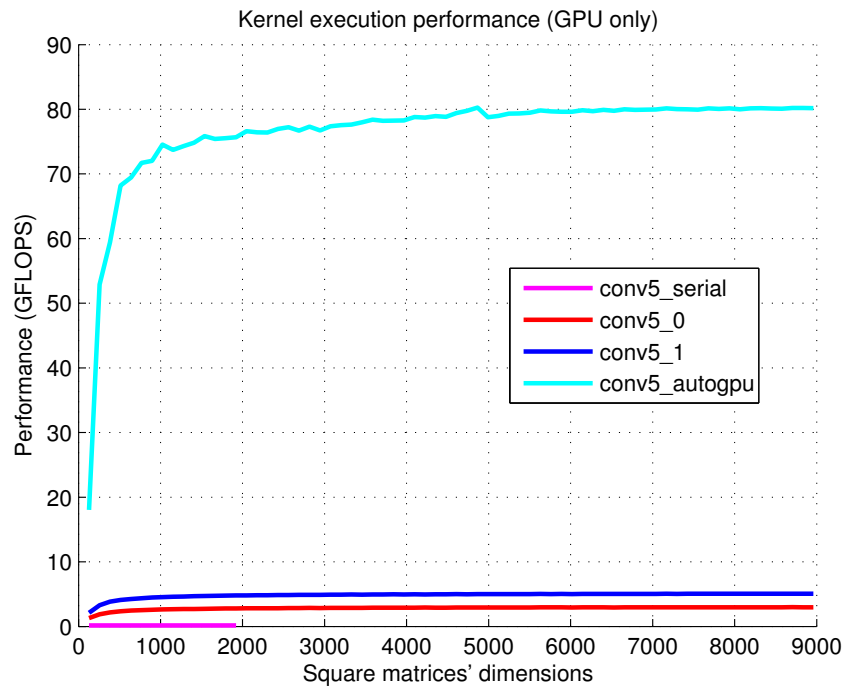


Figure 4.25: Performance of the input, intermediate and final CONV kernels (5×5 template)

Kernel	Execution time	Approximate speed-up		
		to conv5_0	to conv5_1	to conv5_flcc
conv5_0	239 msec	—	—	—
conv5_1	138 msec	42%	—	—
conv5_flcc	49 msec	79%	64%	—
conv5_autogpu	42 msec	82%	70%	14%

Table 4.12: CONV kernel (5×5 template) execution times and relative speed-up for 8192×8192 matrices

Kernel	Performance	Approximate speed-up		
		to conv5_0	to conv5_1	to conv5_flcc
conv5_0	14 GFLOPS	—	—	—
conv5_1	24 GFLOPS	71%	—	—
conv5_flcc	68 GFLOPS	386%	183%	—
conv5_autogpu	80 GFLOPS	471%	234%	18%

Table 4.13: CONV kernel (5×5 template) execution times and relative speed-up for 8192×8192 matrices

the two serial *for* loops may take; also, it is possible that the loop optimization parameters may result in different values than those for the 5×5 template matrix.

The intermediate, optimal and reference kernels that perform the direct convolution of an image with an 11×11 template follow an identical logic as their counterparts that were outlined for convolution of an image with a 5×5 template. Consequently, the source code of these kernels will not be presented herel rather, the relevant measurements and parameters will be presented directly.

The following parallel kernels are discussed:

- conv11_0 : The “vanilla” parallel kernel—a direct translation of the input AST to CUDA source code.
- conv11_1 : The parallel kernel, after staging of partial accumulations.
- conv11_autogpu : The optimized parallel kernel, after unrolling of both parallel and serial loops.
- conv11_flcc : The direct convolution implementation performed by the FLCC library, developed by Georgios Papamakarios and Georgios Rizos.

Optimal and reference kernels’ parameters

The optimal and reference kernels are described by the transformations and parameters summarized in table 4.14.

Kernel	Block size	Staging	__shared__ caching	Loop unroll factors			
				<i>i</i> loop	<i>j</i> loop	<i>ti</i> loop	<i>tj</i> loop
conv11_autogpu	64×1	Yes	No	1	16	1	11
conv11_flcc	16×16	Yes	Yes	1	1	11	11

Table 4.14: Optimal and reference CONV kernels specifications (11×11 template)

Kernel performance

Figures 4.29 and 4.30 show the execution time of and floating-point operations performed by the serial, intermediate and optimal generated direct 2D convolution kernels.

The execution time and performance of the reference and optimal generated kernel can be seen in figures 4.31 and 4.32, for input images of sizes 128×128 to 17280×17280 .

Kernel precision

Table 4.15 lists the maximum absolute error in the convolution matrix computed in the GPU (as compared to that of the CPU), for different input image sizes.

Image size	Maximum absolute error
1024×1024	$4.76836 \cdot 10^{-6}$
2048×2048	$5.72205 \cdot 10^{-6}$
4096×4096	$5.72205 \cdot 10^{-6}$
8192×8192	$5.72205 \cdot 10^{-6}$
16384×16384	$7.62939 \cdot 10^{-6}$

Table 4.15: CONV kernel output's maximum absolute error (11×11 template)

Conclusions

A quantitative comparison of the kernels discussed in this subsection, for an input image matrix of size 8192×8192 , is presented in tables 4.16 and 4.17, regarding execution time and floating-point operations, respectively.

Kernel	Execution time	Approximate speed-up		
		to conv11_0	to conv11_1	to conv11_autogpu
conv5_0	1128 msec	–	–	–
conv5_1	662 msec	41%	–	–
conv5_autogpu	106 msec	91%	84%	–
conv5_flcc	81 msec	93%	88%	24%

Table 4.16: CONV kernel (11×11 template) execution times and relative speed-up for 8192×8192 matrices

Kernel	Performance	Approximate speed-up		
		to conv11_0	to conv11_1	to conv11_autogpu
conv11_0	14 GFLOPS	–	–	–
conv11_1	25 GFLOPS	79%	–	–
conv11_autogpu	153 GFLOPS	993%	512%	–
conv11_flcc	200 GFLOPS	1328%	700%	31%

Table 4.17: CONV kernel (11×11 template) execution times and relative speed-up for 8192×8192 matrices

For an 11×11 template matrix, the optimal kernel generated by the *AutoGPU* program fails to perform better than the FLCC library kernel. The kernels' major differences (see table 4.14) are the specified CUDA blocks and the caching of the image matrix; hence, the inability of the *AutoGPU* program to generate a truly optimal kernel should lie in its not considering 2D CUDA blocks or its not attempting to cache matrices whose access pattern fulfills the coalescing criterion.

Moreover, it should be noted that the kernels' performance stays steady for input image matrices of increasing sizes.

```

__global__ void convolution( float *imageDev, float *templateDev,
                           float *outputDev, int SW, int BW )
{
    const int u0 = IMUL(blockIdx.x, blockDim.x);
    const int v0 = IMUL(blockIdx.y, blockDim.y);

    __shared__ float panelSh[400];
    const int i_idx = u0 + IMUL(v0, BW);
    for (int j=threadIdx.y; j<20; j+=16) {
        #pragma unroll
        for (int i=threadIdx.x; i<20; i+=16) {
            panelSh[IMUL(j, 20) + i] = imageDev[i_idx + IMUL(j, BW) + i];
        }
    }

    __shared__ float templateSh[25];
    const int jtwi = IMUL(threadIdx.y, 5) + threadIdx.x;
    templateSh[jtwi] = templateDev[jtwi];
    __syncthreads();

    register float conv = 0.0;
    const int p_idx = threadIdx.x + IMUL(threadIdx.y, 20);
    conv += panelSh[p_idx + 0] * templateSh[24];
    conv += panelSh[p_idx + 1] * templateSh[23];
    conv += panelSh[p_idx + 2] * templateSh[22];
    conv += panelSh[p_idx + 3] * templateSh[21];
    conv += panelSh[p_idx + 4] * templateSh[20];
    conv += panelSh[p_idx + 20] * templateSh[19];
    conv += panelSh[p_idx + 21] * templateSh[18];
    conv += panelSh[p_idx + 22] * templateSh[17];
    conv += panelSh[p_idx + 23] * templateSh[16];
    conv += panelSh[p_idx + 24] * templateSh[15];
    conv += panelSh[p_idx + 40] * templateSh[14];
    conv += panelSh[p_idx + 41] * templateSh[13];
    conv += panelSh[p_idx + 42] * templateSh[12];
    conv += panelSh[p_idx + 43] * templateSh[11];
    conv += panelSh[p_idx + 44] * templateSh[10];
    conv += panelSh[p_idx + 60] * templateSh[9];
    conv += panelSh[p_idx + 61] * templateSh[8];
    conv += panelSh[p_idx + 62] * templateSh[7];
    conv += panelSh[p_idx + 63] * templateSh[6];
    conv += panelSh[p_idx + 64] * templateSh[5];
    conv += panelSh[p_idx + 80] * templateSh[4];
    conv += panelSh[p_idx + 81] * templateSh[3];
    conv += panelSh[p_idx + 82] * templateSh[2];
    conv += panelSh[p_idx + 83] * templateSh[1];
    conv += panelSh[p_idx + 84] * templateSh[0];
    outputDev[(u0+threadIdx.x) + IMUL((v0+threadIdx.y), SW)] = conv;
}

```

Figure 4.26: conv5_flcc: FLCC library kernel (5×5 template)

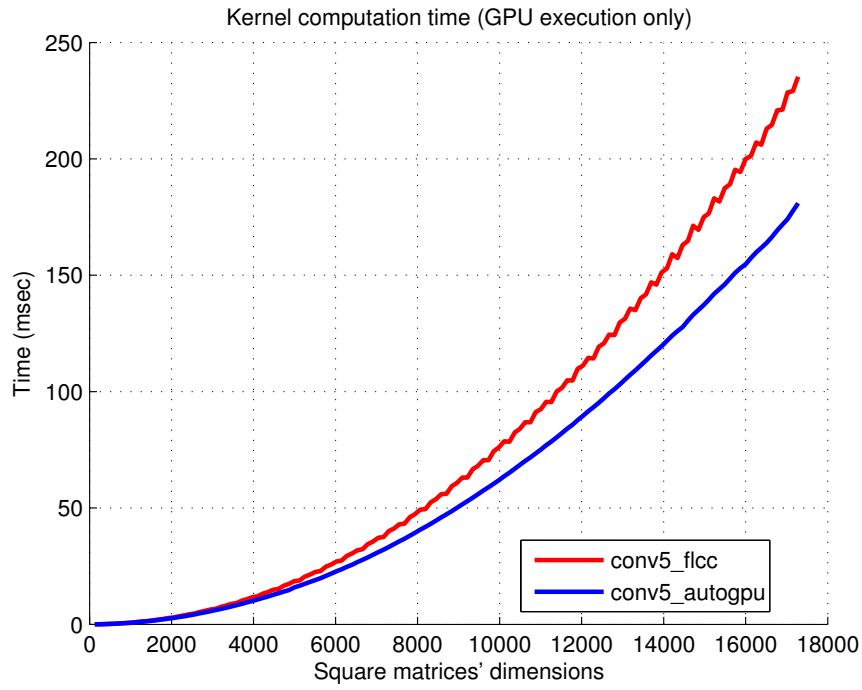


Figure 4.27: Execution times of the the reference and optimal CONV kernels (5×5 template)

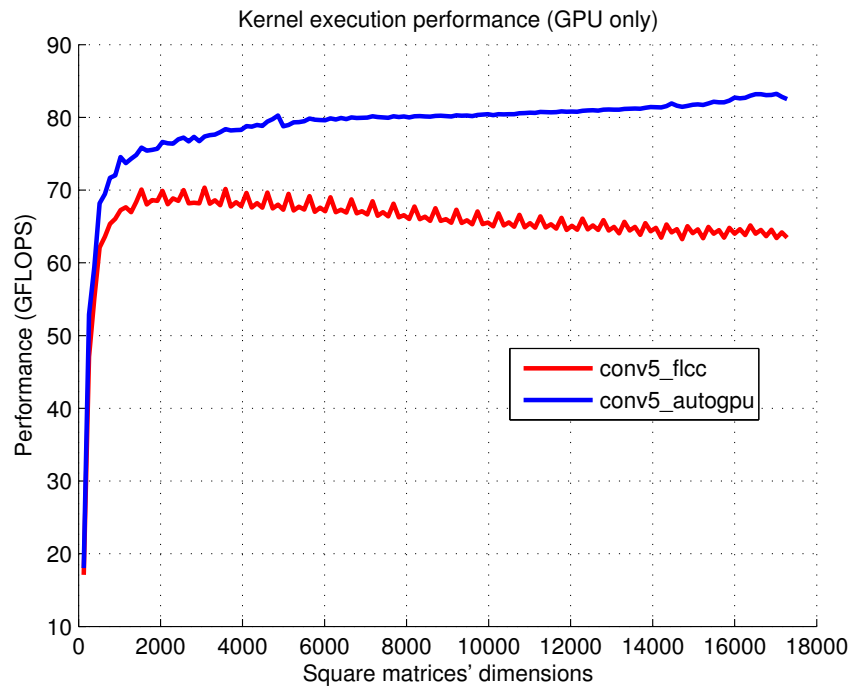


Figure 4.28: Performance of the reference and optimal CONV kernels (5×5 template)

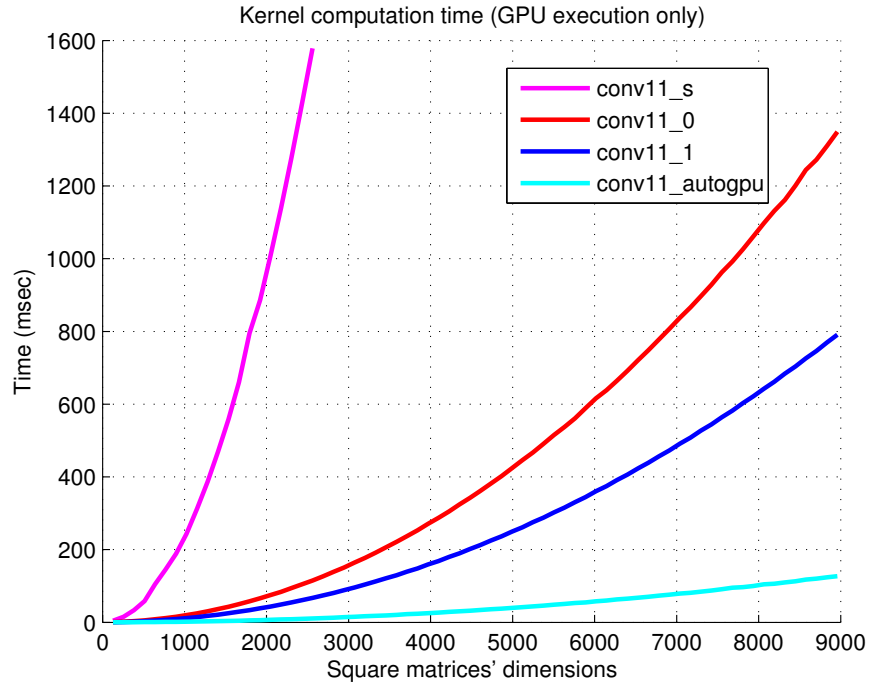


Figure 4.29: Execution times of the input, intermediate and final CONV kernels (11×11 template)

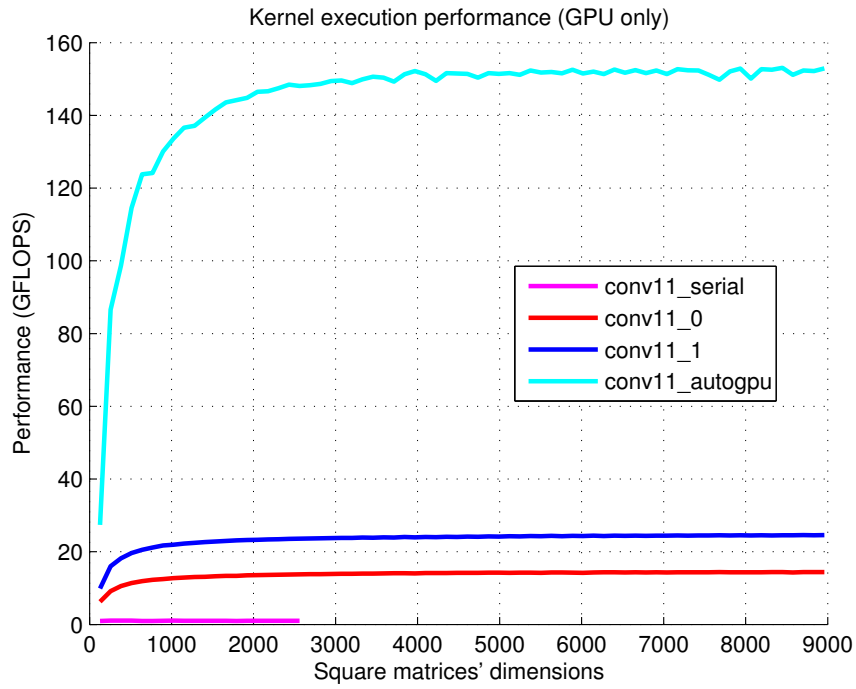


Figure 4.30: Performance of the input, intermediate and final CONV kernels (11×11 template)

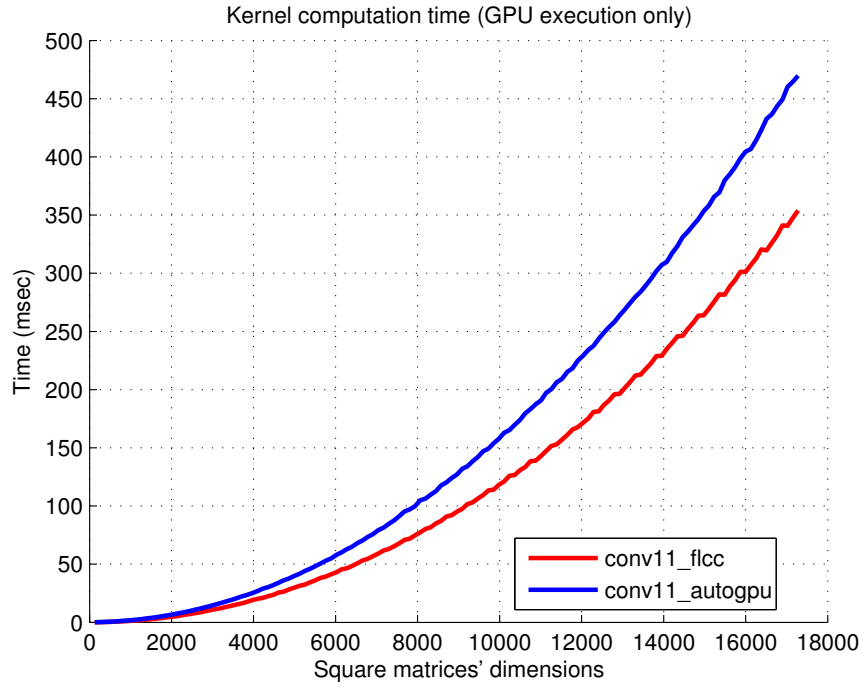


Figure 4.31: Execution times of the reference and optimal CONV kernels (11×11 template)

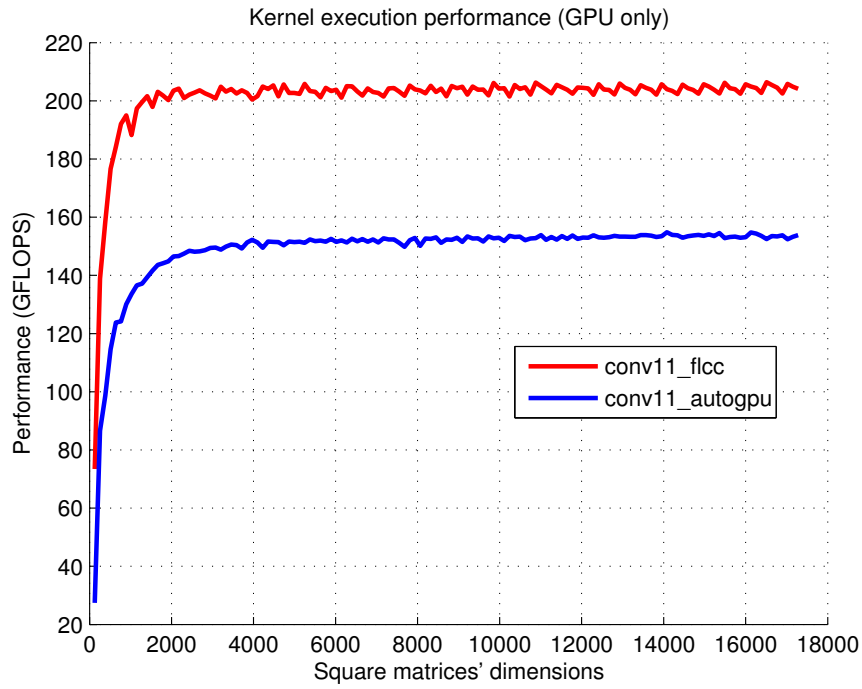


Figure 4.32: Performance of the reference and optimal CONV kernels (11×11 template)

Chapter 5

Conclusions and future considerations

GPUs have advanced to a point that their effectiveness in performing massive computations on a set of data can greatly surpass that of general-purpose CPUs. Exploiting the capabilities of this new tool is definitely worthwhile, especially with regard to possible real-time applications where a program's execution time is a crucial factor.

A new programming model is introduced to allow the utilization of a GPU's resources in solving a computational problem. However, this model adds new difficulties to kernel optimization process: Apart from the "traditional" optimization techniques, such as loop unrolling and strip-mining, the `__global__`, `__shared__` and `__constant__` memory spaces of the CUDA architecture must be accessed in an appropriate pattern; concurrent threads must be synchronized and divergence of the parallel algorithm should be avoided; and a big enough computational load has to be assigned to each computational element in order to hide the memory latency. The experimental results presented in this report give a clearer picture of the performance boost that can be achieved by each of the aforementioned optimizations.

The optimizations that refer to the memory access patterns or to thread synchronization and concurrency may be considered as general good programming practices with respect to the CUDA model. The optimal thread execution configuration of the kernel, on the other hand, is dependent on the architectural characteristics of the GPU the kernel will execute in. Consequently, selecting the appropriate parameters from the plethora of possible ones for a given GPU is important when a high performance is desired.

The *AutoGPU* program attempts to apply all of the above optimizations and performs the search for the optimal kernel configuration. This allows for the generation of a high performance kernel, tweaked for the target GPU, without any (or little) manual optimization. Kernels generated in this fashion were compared with regard to performance against manually optimized kernels that perform the same computational process. The results illustrate the value of the *AutoGPU* program: The generated kernel exhibits a very high performance which may surpass that of the corresponding manually optimized one, without requiring that the user spend many hours, if not days, in its development.

However, there are cases where the *AutoGPU*-generated kernel may fail to perform better than a manually optimized one. This suggests that there are certain limitations in the result that the *AutoGPU* program can achieve. The most significant one is probably the fact that only 1-dimensional CUDA blocks are supported in the generated kernel. Thus, the *AutoGPU* model should be extended to also allow a kernel configuration based on 2- and 3-dimensional CUDA blocks. This would mean an increase in time for the optimization procedure, since for an n -dimensional computation domain, all possible CUDA block configurations, where the block's dimensions are n or lower, must be checked; nonetheless, this is necessary for the generated kernel to be able to achieve a performance as high as possible in all cases.

Furthermore, the *AutoGPU* model does not currently support the optimization of a kernel that refers to a 3-dimensional computation domain. Experiments are needed to deduce the possible

thread-element assignment schemes, as well as support of 3D-pitched arrays, provided after release of CUDA v3.2, for this extension to be realized.

Lastly, to further facilitate the use of the *AutoGPU* program, certain features might be added. An important one would be a C source code parser, since the CUDA language is essentially an extension of the C language, hence an input serial kernel is most likely to be written in C. This, especially coupled with an appropriate user interface, would make the process of providing input to the *AutoGPU* program much more simple.

In conclusion, the CUDA programming and architecture model is constantly evolving, producing new capabilities as well as difficulties for high performance computations. A tool such as *AutoGPU*, whose objective is to take advantage of the model's characteristics in order to automatically produce an optimized CUDA kernel, must be constantly developed along with the CUDA model to achieve peak performance. Nevertheless, the experimental results collected by the current implementation of *AutoGPU* indicate that its development follows the correct direction, and that it may be an invaluable tool in the pursuit of high performance.

Appendices

Appendix A

Examples' testing platforms

In order to optimize a CUDA kernel with the *AutoGPU* program, a testing platform and its input data are needed. The source of the testing platforms of the kernels discussed in chapter 4, as well as of the utility programs that were used to generate input for the kernels are presented in this appendix.

```

#include <iostream>
#include <cstdio>
#include <fstream>
#include <sstream>
#include <cmath>

#include <cuda.h>
#include <cutil.h>

#include "kernel-matmul.cu"

void readInputFile( float*&, float*&, float*&, int&, int&, int&, char* );

int main( int argc, char** argv ) {
    float* A;
    float* B;
    float* C;
    int m;
    int n;
    int p;

    /* Read matrices from input file */
    readInputFile( A, B, C, m, n, p, argv[1] );

    /* Set CUDA memory pointers and allocate space in device memory */
    float* dev_A;
    size_t pitch_dev_A;
    float* dev_B;
    size_t pitch_dev_B;
    float* dev_C;
    size_t pitch_dev_C;
    cudaMallocPitch( (void*)&dev_A, &pitch_dev_A, m * sizeof( float ), n );
    cudaMallocPitch( (void*)&dev_B, &pitch_dev_B, m * sizeof( float ), p );
    cudaMallocPitch( (void*)&dev_C, &pitch_dev_C, p * sizeof( float ), n );

    /* Configure CUDA kernel execution */
    dim3 dimBlock( WIDTH );
    dim3 dimGrid( static_cast<int>( ceil( m / (float)(WIDTH * UNROLL1) ) ),
                  static_cast<int>( ceil( n / (float)UNROLL2 ) ) );

    /* Copy data to CUDA device memory */
    cudaMemcpy2D( dev_B, pitch_dev_B, B, m * sizeof( float ),
                  m * sizeof( float ), p, cudaMemcpyHostToDevice );
    cudaMemcpy2D( dev_C, pitch_dev_C, C, p * sizeof( float ),
                  p * sizeof( float ), n, cudaMemcpyHostToDevice );

    /* Initialize product matrix to 0 */
    cudaMemset2D( dev_A, pitch_dev_A, 0, n * sizeof( float ), m );

    /* Copy pitch of 2D arrays to CUDA constant memory */
    cudaMemcpyToSymbol( pitch_A, &pitch_dev_A, sizeof( pitch_dev_A ), 0,
                        cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( pitch_B, &pitch_dev_B, sizeof( pitch_dev_B ), 0,
                        cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( pitch_C, &pitch_dev_C, sizeof( pitch_dev_C ), 0,
                        cudaMemcpyHostToDevice );

```

```

/* Create timing event objects */
cudaEvent_t tmrStartAGPU;
cudaEvent_t tmrStopAGPU;
cudaEventCreate( &tmrStartAGPU );
cudaEventCreate( &tmrStopAGPU );
float tmrValueAGPU;

/* Tick */
cudaEventRecord( tmrStartAGPU, 0 );

/* RUN THE KERNEL */
matmul<<< dimGrid, dimBlock >>>( dev_A, dev_B, dev_C, m, n, p );

/* Tock */
cudaEventRecord( tmrStopAGPU, 0 );
cudaEventSynchronize( tmrStopAGPU );
cudaEventElapsedTime( &tmrValueAGPU, tmrStartAGPU, tmrStopAGPU );

/* Check for runtime errors */
bool kernelfail = false;
cudaThreadSynchronize();
cuErr = cudaGetLastError();
if ( cuErr != cudaSuccess )
    kernelfail = true;

/* Destroy timing event objects */
cudaEventDestroy( tmrStartAGPU );
cudaEventDestroy( tmrStopAGPU );

/* Save kernel execution time as a string in file 'comptime.agpu' */
std::ofstream outsAGPU( "comptime.agpu", std::ios::out | std::ios::trunc );
std::ostringstream ossTimeAGPU;
ossTimeAGPU << tmrValueAGPU;
std::string strTimeAGPU = ossTimeAGPU.str();
strTimeAGPU.append( "\n" );
outsAGPU.write( strTimeAGPU.c_str(), strTimeAGPU.size() );
outsAGPU.close();

/* Copy result back to host */
cudaMemcpy2D( A, m * sizeof( float ), dev_A, pitch_dev_A,
              m * sizeof( float ), n, cudaMemcpyDeviceToHost );

/* Free resources */
delete A;
delete B;
delete C;
cudaFree( dev_A );
cudaFree( dev_B );
cudaFree( dev_C );

/* Bye-bye */
if ( kernelfail )
    return 1;
else
    return 0;
} // end main

```

```

void readInputFile( float*& A, float*& B, float*& C,
                   int& m, int& n, int& p, char* filename ) {
    /* Open input file stream */
    std::ifstream is( filename );
    if ( !is ) {
        std::cerr << "\nError opening file stream for '" << filename << "'\n\n";
        exit( 2 );
    } // end if

    /* Read matrix dimensions and allocate memory */
    is.read( (char*)&m, sizeof( int ) );
    is.read( (char*)&n, sizeof( int ) );
    is.read( (char*)&p, sizeof( int ) );
    A = new float [m * n];
    B = new float [m * p];
    C = new float [p * n];
    if ( !A || !B || !C ) {
        std::cerr << "\nError allocating memory on host\n\n";
        exit( 3 );
    } // end if

    /* Read file data into host matrices and close the file stream */
    is.read( (char*)B, m * p * sizeof( float ) );
    is.read( (char*)C, p * n * sizeof( float ) );
    is.close();
} // end function 'readInputFile'

```

Figure A.1: The source code of the testing platform for the MATMUL kernel

```

#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
#include <cuda.h>
using namespace std;

#define EL_VAL_WIDTH 2

void genMatrix( float*&, int, int );
void initMatrixZero( float*&, int, int );
void matmul_serial( float*, float*, float*&, int, int, int );

int main( int argc, char* argv[] ) {
    float* A;
    float* B;
    float* product;
    int m;
    int n;
    int p;
    string filename;
    ofstream os;
    int randseed;
    char randChoice;

    /* Read output filename */
    if ( argc == 1 ) {
        cout << "\nInsert output filename: ";
        cin >> filename;
    } // end if
    else {
        filename = argv[1];
    } // end else

    /* Open file stream */
    os.open( filename.c_str() );
    if ( !os ) {
        cerr << "\nError opening file stream for '" << filename << "'\n\n";
        exit( 1 );
    } // end if

    /* Initialize random seed (user-defined or default) */
    if ( argc == 1 ) {
        do {
            cout << "\nDo you want to use your own random seed? (y/n): ";
            cin >> randChoice;
        } while ( (randChoice != 'y' ) && ( randChoice != 'n' ) );
        if ( randChoice == 'y' ) {
            cout << "Insert random seed: ";
            cin >> randseed;
            srand( randseed );
        } // end if
        else {
            cout << "--Default random seed is used\n";
            srand( (long)time( 0 ) );
        } // end else
    } // end if
    else {
        srand( (long)time( 0 ) );
    } // end else
}

```

```

/* Read matrix dimensions */
if ( argc == 1 ) {
    cout << "\nInsert M: ";
    cin >> m;
    cout << "Insert N: ";
    cin >> n;
    cout << "Insert P: ";
    cin >> p;
} // end if
else {
    m = n = p = atoi( argv[2] );
} // end else

/* Allocate memory for matrices and their product */
A = new float [m * p];
B = new float [p * n];
product = new float [m * n];
if ( !A || !B || !product ) {
    cerr << "\nError allocating memory for matrices\n\n";
    exit( 2 );
} // end if

/* Randomly generate matrices A and B */
genMatrix( A, m, p );
genMatrix( B, p, n );

/* Initialize values of product matrix to zero */
initMatrixZero( product, m, n );

/* Tick */
cudaEvent_t tmrStart;
cudaEvent_t tmrStop;
cudaEventCreate( &tmrStart );
cudaEventCreate( &tmrStop );
float tmrValue;
cudaEventRecord( tmrStart, 0 );

/* Compute product of A*B */
matmul_serial( A, B, product, m, n, p );

/* Tock */
cudaEventRecord( tmrStop, 0 );
cudaEventSynchronize( tmrStop );
cudaEventElapsedTime( &tmrValue, tmrStart, tmrStop );
cudaEventDestroy( tmrStart );
cudaEventDestroy( tmrStop );

/* Write dimensions and elements of matrices in output file */
os.write( (char*)&m, sizeof( int ) );
os.write( (char*)&n, sizeof( int ) );
os.write( (char*)&p, sizeof( int ) );
os.write( (char*)A, m * p * sizeof( float ) );
os.write( (char*)B, p * n * sizeof( float ) );
os.write( (char*)product, m * n * sizeof( float ) );

```

```

/* Save execution time as a string in file 'comptime-serial.agpu' */
ofstream outs( "comptime-serial.agpu", ios::out | ios::app );
ostringstream ossTime;
ossTime << tmrValue;
string strTime = ossTime.str();
strTime.append( "\n" );
outs.write( strTime.c_str(), strTime.size() );
outs.close();

/* Farewell */
os.close();
delete A;
delete B;
delete product;
return 0;
} // end main

void genMatrix( float*& mat, int m, int n ) {
    for ( int i = 0; i < m; ++i )
        for ( int j = 0; j < n; ++j )
            mat[i + j*m] = (float)( ((float)rand() / (float)RAND_MAX) - 0.5f)
                             * EL_VAL_WIDTH );
} // end function 'genMatrix'

void initMatrixZero( float*& mat, int m, int n ) {
    for ( int i = 0; i < m; ++i )
        for ( int j = 0; j < n; ++j )
            mat[i + j*m] = 0.0f;
} // end function 'initMatrixZero'

void matmul_serial( float* A, float* B, float*& prod, int m, int n, int p )
    for ( int i = 0; i < m; ++i )
        for ( int j = 0; j < n; ++j )
            for ( int k = 0; k < p; ++k )
                prod[i + j*m] += A[i + k*m] * B[k + j*p];
} // end function 'matmul_serial'

```

Figure A.2: The source code of the input generation utility program for the MATMUL kernel

```

#include <iostream>
#include <cstdio>
#include <fstream>
#include <sstream>
#include <cmath>
#include <cuda.h>
#include <cutil.h>

#include "kernel-convolution.cu"

void readInputFile( float*&, float*&, float*&,
                    int&, int&, int&, int&, int&, int&, char* );

int main( int argc, char** argv ) {
    float* Image;
    float* Out;
    float* Template;
    int IH;
    int IW;
    int TH;
    int TW;
    int sh;
    int sw;
    int SH;
    int SW;
    int BH;
    int BW;

    /* Read matrices from input file */
    readInputFile( Out, Image, Template, IH, IW, TH, TW, sh, sw, argv[1] );

    /* Calculate 'extended' padded dimensions of output */
    SH = sh + (((WIDTH*UNROLL1) - (sh % (WIDTH*UNROLL1))) % (WIDTH*UNROLL1));
    SW = sw + ((UNROLL2 - (sw % UNROLL2)) % UNROLL2);
    BH = SH + TH - 1;
    BW = SW + TW - 1;

    /* Configure CUDA kernel execution */
    dim3 dimBlock( WIDTH );
    dim3 dimGrid( SH / (WIDTH*UNROLL1), SW / UNROLL2 );

    /* Set CUDA memory pointers and allocate space in device memory */
    float* dev_O;
    size_t pitch_dev_O;
    float* dev_I;
    size_t pitch_dev_I;
    cudaMallocPitch( (void**)&dev_O, &pitch_dev_O, SH * sizeof( float ), SW );
    cudaMallocPitch( (void**)&dev_I, &pitch_dev_I, BH * sizeof( float ), BW );

    /* Copy pitch of 2D arrays to CUDA constant memory */
    cudaMemcpyToSymbol( pitch_O, &pitch_dev_O, sizeof( pitch_dev_O ), 0,
                        cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( pitch_I, &pitch_dev_I, sizeof( pitch_dev_I ), 0,
                        cudaMemcpyHostToDevice );

    /* Copy template and output dimensions to constant memory */
    cudaMemcpyToSymbol( T_H, &TH, sizeof( TH ), 0, cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( T_W, &TW, sizeof( TW ), 0, cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( T_Template, TH * TW * sizeof( float ), 0,
                        cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( S_H, &SH, sizeof( SH ), 0, cudaMemcpyHostToDevice );
    cudaMemcpyToSymbol( S_W, &SW, sizeof( SW ), 0, cudaMemcpyHostToDevice );

```

```

/* Initialize image and output to zeros */
cudaMemset2D( dev_O, pitch_dev_O, 0, SH * sizeof( float ), SW );
cudaMemset2D( dev_I, pitch_dev_I, 0, BH * sizeof( float ), BW );

/* Copy image to CUDA device memory */
cudaMemcpy2D( (float*)( (char*)dev_I + (TW-1) * pitch_dev_I ) + (TH-
    pitch_dev_I,
    Image, IH * sizeof( float ), IH * sizeof( float ), IW,
    CudaMemcpyHostToDevice );

/* Create timing event objects */
cudaEvent_t tmrStartAGPU;
cudaEvent_t tmrStopAGPU;
cudaEventCreate( &tmrStartAGPU );
cudaEventCreate( &tmrStopAGPU );
float tmrValueAGPU;

/* Tick */
cudaEventRecord( tmrStartAGPU, 0 );

/* RUN THE KERNEL */
convolution<<< dimGrid, dimBlock >>>( dev_O, dev_I );

/* Tock */
cudaEventRecord( tmrStopAGPU, 0 );
cudaEventSynchronize( tmrStopAGPU );
cudaEventElapsedTime( &tmrValueAGPU, tmrStartAGPU, tmrStopAGPU );

/* Check for runtime errors */
bool kernelfail = false;
cudaThreadSynchronize();
cudaError_t cuErr = cudaGetLastError();
if ( cuErr != cudaSuccess )
    kernelfail = true;

/* Destroy timing event objects */
cudaEventDestroy( tmrStartAGPU );
cudaEventDestroy( tmrStopAGPU );

/* Save kernel execution time as a string in file 'comptime.agpu' */
std::ofstream outsAGPU( "comptime.agpu", std::ios::out | std::ios::trunc )
std::ostringstream ossTimeAGPU;
ossTimeAGPU << tmrValueAGPU;
std::string strTimeAGPU = ossTimeAGPU.str();
strTimeAGPU.append( "\n" );
outsAGPU.write( strTimeAGPU.c_str(), strTimeAGPU.size() );
outsAGPU.close();

/* Copy result back to host */
cudaMemcpy2D( Out, sh * sizeof( float ), dev_O, pitch_dev_O,
    sh * sizeof( float ), sw, cudaMemcpyDeviceToHost );

/* Free resources */
delete Out;
delete Image;
delete Template;
cudaFree( dev_O );
cudaFree( dev_I );

```

```

/* Bye-bye */
if ( kernelfail )
    return 1;
else
    return 0;
} // end main

void readInputFile( float*& Out, float*& Image, float*& Template,
                   int& IH, int& IW, int& TH, int& TW, int& sh, int& sw,
                   char* filename ) {
    /* Open input file stream */
    std::ifstream is( filename );
    if ( !is ) {
        std::cerr << "\nError opening file stream for '" << filename << "'\n\n";
        exit( 2 );
    } // end if

    /* Read matrix dimensions and allocate memory */
    is.read( (char*)&IH, sizeof( int ) );
    is.read( (char*)&IW, sizeof( int ) );
    is.read( (char*)&TH, sizeof( int ) );
    is.read( (char*)&TW, sizeof( int ) );
    is.read( (char*)&sh, sizeof( int ) );
    is.read( (char*)&sw, sizeof( int ) );
    Out = new float [sh * sw];
    Image = new float [IH * IW];
    Template = new float [TH * TW];
    if ( !Out || !Image || !Template ) {
        std::cerr << "\nError allocating memory on host\n\n";
        delete Out;
        delete Image;
        delete Template;
        is.close();
        exit( 3 );
    } // end if

    /* Read file data into host matrices and close the file stream */
    is.read( (char*)Image, IH * IW * sizeof( float ) );
    is.read( (char*)Template, TH * TW * sizeof( float ) );
    is.close();
} // end function 'readInputFile'

```

Figure A.3: The source code of the testing platform for the CONV kernel

```

#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <cuda.h>
using namespace std;

#define EL_VAL_WIDTH 2

void genMatrix( float*, int, int );
void initMatrixZero( float*&, int, int );
void convolution_serial( float*, float*, float*&,
                       int, int, int, int, int, int, int );

int main( int argc, char* argv[] ) {
    float* Image;
    float* Template;
    float* result;
    int IH;
    int IW;
    int TH;
    int TW;
    int sh;
    int sw;
    int BH;
    int BW;
    string filename;
    ofstream os;
    int randseed;
    char randChoice;

    /* Read output filename */
    if ( argc == 1 ) {
        cout << "\nInsert output filename: ";
        cin >> filename;
    } // end if
    else {
        filename = argv[1];
    } // end else

    /* Open file stream */
    os.open( filename.c_str() );
    if ( !os ) {
        cerr << "\nError opening file stream for '" << filename << "'\n\n";
        exit( 1 );
    } // end if

    /* Initialize random seed (user-defined or default) */
    if ( argc == 1 ) {
        do {
            cout << "\nDo you want to use your own random seed? (y/n): ";
            cin >> randChoice;
        } while ( (randChoice != 'y' ) && ( randChoice != 'n' ) );
        if ( randChoice == 'y' ) {
            cout << "Insert random seed: ";
            cin >> randseed;
            srand( randseed );
        } // end if
    }
}

```

```

        else {
            cout << "--Default random seed is used\n";
            srand( (long)time( 0 ) );
        } // end else
    } // end if
    else {
        srand( (long)time( 0 ) );
    } // end else

    /* Read matrix dimensions */
    if ( argc == 1 ) {
        cout << "\nInsert # of rows for Input matrix: ";
        cin >> IH;
        cout << "Insert # of columns for Input matrix: ";
        cin >> IW;
        cout << "\nInsert # of rows for Template matrix: ";
        cin >> TH;
        cout << "Insert # of columns for Template matrix: ";
        cin >> TW;
    } // end if
    else {
        IH = IW = atoi( argv[2] );
        TH = TW = atoi( argv[3] );
    } // end else

    /* Calculate output and padded input dimensions */
    sh = IH + TH - 1;
    sw = IW + TW - 1;
    BH = sh + TH - 1;
    BW = sw + TW - 1;

    /* Allocate memory for image, template and convolution result */
    Image = new float [BH * BW];
    Template = new float [TH * TW];
    result = new float [sh * sw];
    if ( !Image || !Template || !result ) {
        cerr << "\nError allocating memory for matrices\n\n";
        delete Image;
        delete Template;
        delete result;
        os.close();
        exit( 2 );
    } // end if

    /* Initialize convolution result and image (padded) to zeros */
    initMatrixZero( result, sh, sw ); //!
    initMatrixZero( Image, BH, BW );

    /* Randomly generate image and template */
    for ( int j = TW - 1; j < IW + TW - 1; ++j )
        genMatrix( Image + (TH-1) + j*BH, IH, 1 );
    genMatrix( Template, TH, TW );

    /* Tick */
    cudaEvent_t tmrStart;
    cudaEvent_t tmrStop;
    cudaEventCreate( &tmrStart );
    cudaEventCreate( &tmrStop );
    float tmrValue;
    cudaEventRecord( tmrStart, 0 );

```

```

/* Compute 2D convolution of (Image ** Template) */
convolution_serial( Image, Template, result,
                    IH, IW, TH, TW, sh, sw, BH, BW ); //!

/* Tock */
cudaEventRecord( tmrStop, 0 );
cudaEventSynchronize( tmrStop );
cudaEventElapsedTime( &tmrValue, tmrStart, tmrStop );
cudaEventDestroy( tmrStart );
cudaEventDestroy( tmrStop );

/* Write dimensions and elements of matrices in output file */
os.write( (char*)&IH, sizeof( int ) );
os.write( (char*)&IW, sizeof( int ) );
os.write( (char*)&TH, sizeof( int ) );
os.write( (char*)&TW, sizeof( int ) );
os.write( (char*)&sh, sizeof( int ) );
os.write( (char*)&sw, sizeof( int ) );
for ( int j = TW - 1; j < IW + TW - 1; ++j )
    os.write( (char*)(Image + (TH-1) + j*BH), IH * sizeof( float ) );
os.write( (char*)Template, TH * TW * sizeof( float ) );
os.write( (char*)result, sh * sw * sizeof( float ) );

/* Save execution time as a string in file 'comptime-serial.agpu' */
ofstream outs( "comptime-serial.agpu", ios::out | ios::app );
ostringstream ossTime;
ossTime << tmrValue;
string strTime = ossTime.str();
strTime.append( "\n" );
outs.write( strTime.c_str(), strTime.size() );
outs.close();

/* Farewell */
os.close();
delete Image;
delete Template;
delete result;
return 0;
} // end main

void genMatrix( float* mat, int m, int n ) {
    for ( int i = 0; i < m; ++i )
        for ( int j = 0; j < n; ++j )
            mat[i + j*m] = (float)( (((float)rand() / (float)RAND_MAX) - 0.5f)
                                   * EL_VAL_WIDTH );
} // end function 'genMatrix'

void initMatrixZero( float*& mat, int m, int n ) {
    for ( int i = 0; i < m; ++i )
        for ( int j = 0; j < n; ++j )
            mat[i + j*m] = 0.0f;
} // end function 'initMatrixZero'

```

```

void convolution_serial( float* Image, float* Template, float*& Out,
                        int IH, int IW, int TH, int TW,
                        int SH, int SW, int BH, int BW )
{
    for ( int i = 0; i < SH; ++i ) {
        for ( int j = 0; j < SW; ++j ) {

            for ( int ti = 0; ti < TH; ++ti ) {
                for ( int tj = 0; tj < TW; ++tj ) {
                    Out[i + j*SH] += Image[(i+ti) + (j+tj)*BH]
                                   * Template[(TH-1-ti) + (TW-1-tj)*TH];
                } // end for (counter tj)
            } // end for (counter ti)

        } // end for (counter j)
    } // end for (counter i)
} // end function 'convolution_serial'

```

Figure A.4: The source code of the input generation utility program for the CONV kernel

Bibliography

- [1] Lawrence C. Paulson, 1996. *ML for the Working Programmer*, 2nd Edition, Cambridge University Press.
- [2] NVIDIA Corporation, 2010. *NVIDIA CUDATM: NVIDIA CUDA C Programming Guide, version 3.2*, NVIDIA Corporation. [\[Link\]](#)
- [3] Michael Wolfe, 2008. *Compilers and More: Optimizing GPU Kernels*, HPCwire. [\[Link\]](#)
- [4] Andrew W. Appel, 2004. *Modern Compiler Implementation in ML*, Cambridge University Press.
- [5] Georgios Papamakarios, Georgios Rizos, 2011. *FLCC Library: A Tool for Fast Computation of Convolution and Correlation Coefficients of 2D and 3D Images*, Diploma Dissertation, Aristotle University of Thessaloniki, Dept. of Electrical & Computer Engineering.