J Supercomput (2016) 72:468–502
DOI 10.1007/s11227-015-1575-9

# Tuning framework for stencil computation in heterogeneous parallel platforms

**Taieb Lamine Ben Cheikh**[1] · **Alexandra Aguiar**[1] · **Sofiene Tahar**[2] · **Gabriela Nicolescu**[1]

Published online: 30 November 2015


**Abstract** Image processing and computer vision applications are usually complex in terms of the large amount of processed data and high computation loads. To cope with this, optimization techniques and high-performance hardware platforms are required. Since these applications present many opportunities for parallelism, heterogeneous parallel platforms (HPPs) are an interesting choice, offering a good balance between high computation capabilities and flexibility to handle a large spectrum of application features. Applications such as image filtering and edge detection make extensive use of finite difference method to solve partial derivative equations, which computational pattern is called *stencil computation*. Stencil computations are known as memory-bound, so that reducing high-latency memory access becomes the biggest challenge to reach high performance. In this paper, we present our methodology as a basis of a performance tuning framework to optimize the implementation of multiple stencil computations on HPPs. Results show that our approach outperforms SDK-based methodologies, improving performance. Moreover, using the proposed approach, the developer has the ability of investigating efficiently the performance of the stencil computations before implementing actual code on the target platforms.

✉ Gabriela Nicolescu
gabriela.nicolescu@polymtl.ca

Taieb Lamine Ben Cheikh
taieb.lamine-ibnecheikh@polymtl.ca

Alexandra Aguiar
alexandra.aguiar@polymtl.ca

Sofiene Tahar
tahar@encs.concordia.ca

1 Ecole Polytechnique de Montréal, Montréal, Canada

2 Concordia University, Montréal, Canada

**Keywords** Stencil computation · Computer vision · Heterogeneous parallel platforms

## 1 Introduction

Modern applications, such as image processing and computer vision, are becoming more and more complex [9]. This complexity implies a large amount of processed data and high computation loads, under very tight constraints, such as real-time execution and power consumption awareness. To cope with this, optimization techniques and high-performance hardware platforms are required.

Parallelizing computationally intensive imaging tasks is not a new topic per se [1–4]. However, considering the domains of image processing and computer vision, it is safe to say that they present many opportunities for parallelism that can be exploited by parallel platforms. These platforms may be found in high-performance computers (HPC) or embedded computers. Recently, both HPC and embedded computers are moving toward a heterogeneous computing. They are employing both central processing units (CPUs) and graphics processing units (GPUs) to achieve highest performance. For instance, the supercomputer code-named Titan uses almost 300,000 CPU cores and up to 18,000 GPU cards for processing purposes [6].

Thus, heterogeneous parallel platforms (HPPs) are an interesting choice that can respect the above-mentioned constraints. They offer a good balance between high computation capabilities and flexibility to handle large spectrum of applications features [16]. Most of these platforms adopt a host–device model and, usually, the host is a cache-based heavy multi-core CPU while the device is essentially a scratchpad memory-based light many-core CPU or GPU.

Under the domain of image processing a broad spectrum of computation can be done including pattern recognition for several purposes [13,17]. Still, applications such as image filtering and edge detection make extensive use of the finite difference method to solve partial derivative equations (PDEs). The computational pattern which expresses the finite difference method is called *stencil computation* [8]. In stencil computation, each point of the computed space is updated depending on the points at its neighborhood, which is called *stencil*. Figure 1 depicts a representation of a stencil (*updated element*, in the figure) surrounded by its neighbors (*neighbor elements*, in the figure).

In order to implement computer vision and image processing applications on HPPs, a number of parameters have to be considered. Stencil computations are known as memory-bound computation, so that reducing high-latency memory access becomes the biggest challenge to reach high performance.

To overcome this issue, it is needed to ensure better data locality through the efficient use of low-latency scratchpad shared memory [11]. However, this is not obvious for the developer, especially due to the existence of:

1. a complex application data access pattern, and
2. resource constraints, such as the available low-latency memory space and the maximum number of executed threads.

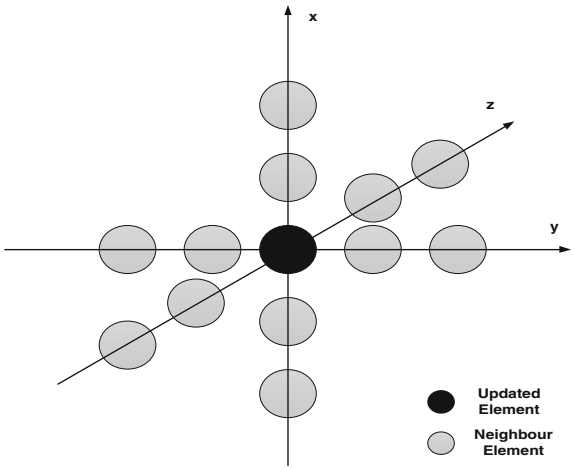


**Fig. 1** Stencil representation

Therefore, a tuning framework is mandatory to guide the developer to reach high performance while respecting those constraints.

In this paper, we present our methodology as a basis of a performance tuning framework to optimize the implementation of multiple stencil computations on HPPs. In this context, the main contribution of this paper is the definition and development of a new performance tuning framework for stencil computation targeting HPPs. We provide a framework tool that takes into account characteristics of both the application and the target architecture while considering multiple stencils computation, which allows more complex applications to be optimized using our framework. The emphasis is put on HPP platforms that follow host–device architectural model where the host corresponds to a multi-core CPU and the device corresponds to a many-core GPU.

Also, the framework is based on a step-wise methodology that helps the developer, who seeks to optimize the stencil implementation on GPU-based platforms, to answer these three main questions:

1. What are the stencils that may be fused to reduce access to high-latency memory by keeping the effective data in the low-latency memory as long as possible?
2. How to find the appropriate tile size and tile shape that may fit with the low-latency memory size and which provides optimal performances?
3. What is the best mapping of the computation load onto the threads hierarchy (grids and blocks of threads)?

The rest of the paper is organized as follows. Section 2 presents more details on stencil computations and strategies for their parallelization. Section 3 surveys the state-of-the-art on improving the implementation of stencil computation on parallel platforms. Section 4 describes in detail our proposed performance tuning framework for stencil computation running on GPU-based HPP platforms. Section 5 gives the experimental results applied on two image processing applications and provides comparison between a basic implementation and an improved one using the proposed framework. Finally, Sect. 6 concludes this paper presenting a summary of the achieved work.

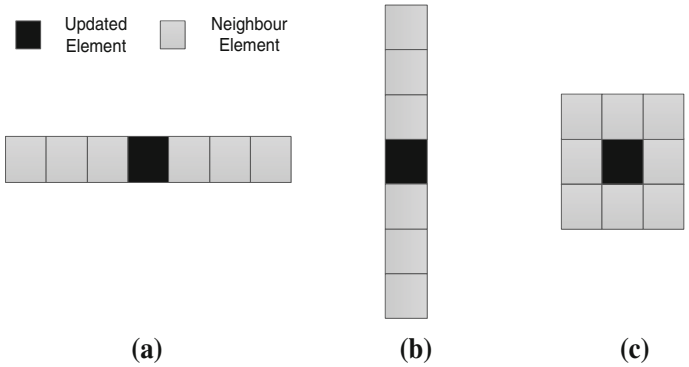


**Fig. 2** Possibilities of stencil shapes. **a** Horizontal 1D stencil. **b** Vertical 1D stencil. **c** 2D stencil

## 2 Basic concepts

This section presents basic concepts regarding stencil computation and its implementation strategies for parallel platforms. The reader who is familiar with these concepts may feel comfortable in going directly to Sect. 3, where we show state-of-the art studies.

### 2.1 Stencil computation

As mentioned before, the stencil computation is the computational pattern that expresses the finite difference method. The stencil itself may be defined by three properties:

1. *stencil operations*, that is, the type of operations performed on neighbor points to update the central point;
2. *stencil shape*, that is, the topology of the neighbor points that are unchanged during the computation. Figure 2 depicts three possible situations regarding horizontal and vertical 1D, and 2D computations (namely *a*, *b* and *c* in the figure), and
3. *stencil space*, that is, the space of points that need to be updated. In general, this space is a regular structured multi-dimensional grid.

Also, the stencil computation may take several forms, such as:

- *single-stencil computation*, where a single stencil is applied only once on the stencil space. For example, we have image convolution on rows;
- *multiple stencils computation*, where a sequence of different stencils is applied on the stencil space. The Canny edge detection is an example of this form, and
- *iterative stencils computation*, where one or several stencils are applied on the stencil space repeatedly and the number of iterations may be known at compile time. In this case, we have simulations with fixed number of time steps, or unknown at compile time that depend on a certain condition. As example we have image shape refining.

It is important to highlight that, in this work, we focus on the *multiple stencils computation*. However, our approach may also be applied for iterative stencils computations.

### 2.2 Parallelization strategies for stencil computation

When implementing stencil computations on GPU-based HPP platforms, many considerations have to be taken into account. They vary according to the complexity of the target platform and the particular properties of stencil computations. These computations are known to be memory bound showing a low compute intensity (the ratio of computing operations by the memory accesses). Thus, the main emphasis is put on improving data access time by improving the data locality. This is guaranteed by the use of low-latency memory, such as scratchpad shared memory, as much as possible. Two techniques are proposed in literature to improve the data locality: *spatial tiling* and *temporal tiling* [8].

Stencil computations are applied on a large amount data that usually surpasses the number of cores, and even the number of threads available on the target hardware platforms. As a consequence, to achieve an improved implementation with optimal locality, the *spatial tiling* technique is commonly used.

The *tiling technique* consists in splitting the data or the stencil space into tiles. Each tile is then processed by one or by a group of threads. In a GPU-based platform, each tile is loaded once in the shared memory. Then it is processed iteratively by a group of threads sharing this tile. This makes the data access to each element of the tile very fast.

However, by dividing the data into tiles, the *border dependency* problem arises. The elements at the borders of each tile need their neighbors to be updated. Since these neighbors are belonging to a different tile, the access to this data will imply in a considerable overhead due to extra communications and synchronizations.

To overcome this overhead, an approach named *overlapped tiling* [12] is adopted. This approach consists in augmenting each tile with extra elements called *halos*, which are loaded and re-computed twice in two neighbor tiles according to one dimension. Nevertheless, this implies in re-loading and re-computing overhead that are proportional to the number of halos and their sizes.

In this case, the biggest challenge is to reduce the number of halos by choosing the appropriate tile layout. An example illustrated in Fig. 3 shows that depending on the tiling, both the number and the size of needed halos will change. The tiling of the stencil space shown in Fig. 3a may follow two different options: the *tiling 1*, depicted in Fig. 3b or the *tiling 2*, shown in Fig. 3c, where the halos are in gray color. It is possible to observe that the *tiling 1* involves a larger number of halos than *tiling 2*. In *tiling 1* we have overlapped tiling (represented with dashed lines in the figure), since the dependencies are involved at both left and right sides of the tile.

Apart from spatial tiling, the *temporal tiling* technique is used for the iterative and multiple stencil computations. Since they involve many intermediate data, we need to increase the data reuse. In a naive implementation, each time we update an intermediate data, we would access the high-latency memory (device memory). In an optimized

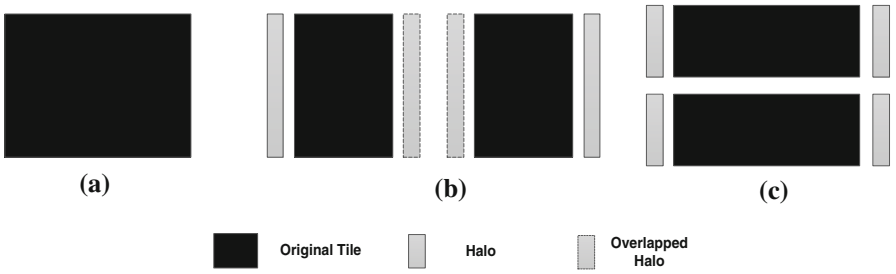


**Fig. 3** Tiling variants. **a** Original stencil space. **b** Tiling 1. **c** Tiling 2

implementation, we will keep the intermediate data in the low-latency memory for as long as they are needed by the computation.

In a GPU implementation, each single-stencil computation is usually implemented as a separate kernel.[1] Since the shared memory does not preserve the data between two kernel launches, multiple stencils have to be fused in one stencil to be launched only once in the case of temporal tiling.

However, by fusing multiple stencils, we have to deal with dependency at borders. To do that, we need to load the appropriate halos and find the number of allocated data and their size. Nevertheless, this definition depends on their lifetimes during the fused stencil computation.

A non-optimized fusion may affect significantly the performance when the shared memory is inefficiently used. Also, performance is affected if both loading and re-computing overheads become more significant when compared to the efficient computation. Moreover, by allocating a large-sized data in the shared memory, the occupancy rate[2] will be reduced, as will the concurrency degree.

## 3 Related work

Stencil computation is a well-known class of computations used in various domains ranging from physics simulations to image processing. A lot of effort has been spent in improving both its productivity and its performance for a wide range of target hardware platforms and, in particular, GPU-based platforms. This effort takes various forms, such as:

1. *improving stencil computation runtime*, by elaborating algorithmic and coding optimizations that may be implemented in specific compilers or as templates in specialized libraries;
2. *performance tuning tools*, by developing tuning tools, and/or;

[1] A kernel is a function that runs on a GPU. One kernel is executed at a time and many threads execute each kernel.

[2] Occupancy rate is defined as the ratio of the number of allocated threads by the limit allowed by each streaming multiprocessor (SM).

3. *evaluation of stencil computation implementations*, by elaborating specialized high-level frameworks based on the definition of new domain-specific languages (DSL) for stencil computations.

Following is a brief discussion of related work classified according to the these mentioned forms.

### 3.1 Related work on improving stencil computation runtime

Several work improve the stencil computation runtime by proposing a number of algorithmic and coding optimizations such as cache-oblivious algorithms, space and temporal blocking via tiling, overlapped tiling and register blocking. The work presented in [20] proposes a set of code optimizations implemented in a domain-specific compiler called *Pochoir*. Many approaches have focused on different levels of blocking to improve cache locality on both multi-core CPU and GPU architectures. Datta et al. studied and evaluated several optimization such as array padding, multi-level blocking, loop unrolling and reordering for stencil computation on a wide variety of hardware architectures [8]. In [12], the authors propose an automatic parallelization of stencil computations that aims to execute tiles in a load balanced manner. For such they propose the use of overlapped tiling that aims to eliminate inter-tile dependencies. Such approach is effective and is employed in our framework. In [11] the authors propose the use of split tiling for GPUs, also aiming automatic parallelization. In this case, the main idea is to avoid redundant computations that are introduced by the overlapped tiling technique, mentioned above. In [19], the authors explore different data access strategies to improve the performance of stencil kernels. They propose an algorithm that aims to optimize the determination of the boundary data elements (halo regions). Unlike the automatic parallelization approach, which is known as a conservative approach, our work is based on a framework that offers the possibility to employ deep optimization techniques as overlapped tiling and padding in order to reach highly tuned implementations of multiple stencil computation.

### 3.2 Related work on performance tuning tools

There are studies that focus in developing tuning tools that target the optimization of one or more of these metrics: the tiling size, the halo size, the compute intensity, and the thread blocks' size for GPUs.

Meng et al. developed an analytical performance model that automatically selects the halo size, which gives the best speedup on GPUs [15]. However, this model is limited to iterative stencil loops where only one stencil is involved in the whole program and, thus, not applicable to more complex code involving multiple stencils. On the other hand, Refs. [18,21] deal with complex iterative multiple stencils where they study the impact of several combinations of kernels fusion/fission on compute intensity for GPUs via an exhaustive search. However they do not provide an analytical model to guide the search process. In [22] the authors show how tiling can be used for perfor-

mance tuning on different models of GPUs. For such the authors implement different tiling strategies aiming to prove that it is hard to achieve a generic tiling strategy that suits every GPU model. Examples of features that can interfere in the tiling technique performance are: number of registers per shared memory (SM), active threads per SM, total scratchpad memory, global memory size among others. When comparing to our work, our main contribution against these studies is to define a complete tuning framework based on an analytical model to optimize the implementation of multiple stencils computation.

### 3.3 Evaluation of stencil computation implementations

Some other contributions are focused on evaluating the suitability of target platforms while running stencil computations. Specific compute capabilities of each platform were investigated and some effective implementation strategies were developed to deliver the best possible performances. In [10], the authors optimize the implementation of stencil computations on accelerated processing units (APU) by proposing a hybrid approach. The main idea is to assign different parts of the stencil computation to different APU processors. Since border treatments are source of high cost of computation and memory divergence, they are assigned to the CPU, whereas the regular computation is assigned to the GPU part. The authors in [5] compare the computation performance of two successive families of both discrete and integrated GPUs while running stencil code. In their paper, they evaluate the impact of PCI express[3] on performance levels. In the same direction, the authors in [14] evaluate the impact of PCI express on performances in the case of multi-GPU platform. Such studies have guided us towards the implementation of a GPU-based solution.

### 3.4 Positioning our approach

In summary, several approaches have been previously proposed for improving stencil computation performance and productivity. Complier-based approaches propose conservative tuning performance. On the other hand, implementation evaluations propose some explorations but they are not based on well-defined tuning frameworks. Finally, few approaches propose tuning frameworks, but are limited to iterative stencil computation. Comparing with these contributions, the main strengths of the proposed approach are:

1. as opposed to conservative approaches, our framework is based on an *analytical model* taking into account the characteristics of the application as well as the characteristics of the targeted architecture, and
2. by considering *multiple stencils computation*, more complex applications may be optimized using our framework.

[3] Peripheral component interconnect express.

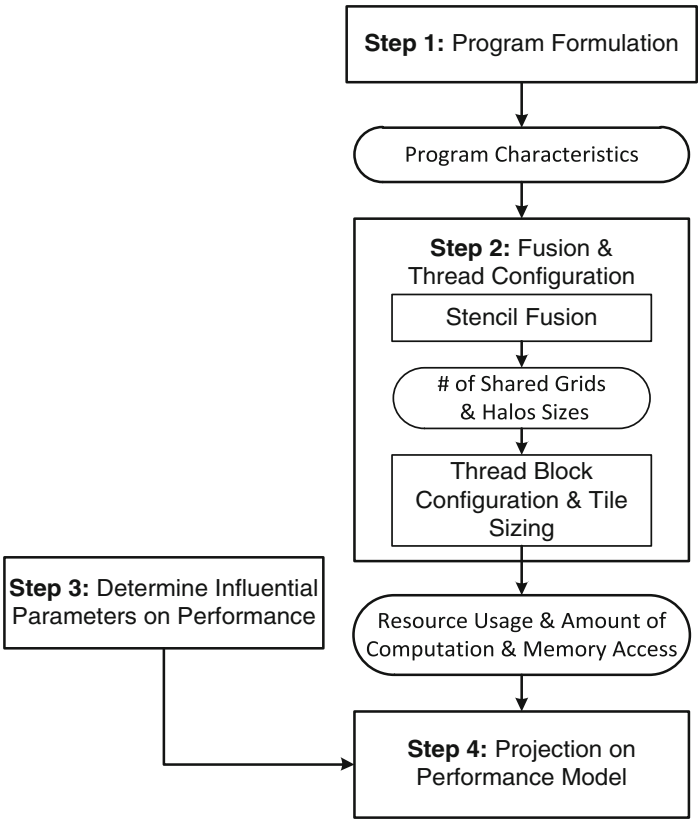


**Fig. 4** Proposed methodology

## 4 The proposed framework

In this section, we describe our proposed performance tuning framework specifically designed for stencil computation running on a GPU-based HPP platform. The framework takes into account characteristics of both the application and the target architecture. Still, by allowing multiple stencil computations, the proposed framework can be used by a large set of applications. Our methodology to build the proposed framework follows four basic steps, depicted in Fig. 4.

The methodology steps are:

- *Step 1: formulation for stencil computation running on HPP* This formulation allows abstracting implementation details while keeping the relevant aspects that have an impact on performance. The high-level representation can then be projected on a performance model to predict the performance. Thus, the developer does not have to implement a full functional code in CUDA or OpenCL for every parallel version to check its performance. Details about the formulation and illustrative examples of high-level representations are provided in Sect. 4.1.

- *Step 2: generates all possible parameters' configurations depending on the target platform* For each configuration, we need to define parameters such as the resource usage (shared memory usage, thread occupancy and the number of resident blocks) and the number of iterations needed to compute the full stencil space. The outputs of this step are used later to select the set of optimal configurations with best performances. Details about this step are provided in Sect. 4.2.
- *Step 3: determines the influential parameters on performance metrics and identifies those that can be controlled by the developer to tune performance* For this purpose, we profile three data access patterns used in three stencil shapes, which are the basic stencils used in a majority of image processing applications. Details about this step are provided in Sect. 4.3.
- *Step 4: we provide at this step a performance model for stencil computation* This model takes as input the resource usage and the computation load provided by step 2, and the impact of a set of controlled parameters on performance metrics that are provided by step 3. This performance model allows to assign to each implementation configuration a computational cost that is used to determine the highest performance configurations. Details about this step are provided in Sect. 4.5.

## 4.1 Formulation of stencil computation running on HPP (step 1)

In this section, we define the first step of our methodology. We provide a formulation for a stencil-based program that runs on an HPP based on a host–device schema. We analyze the particular case of a CPU–GPU platform, as depicted in Fig. 5.

To achieve our goal, we first define a formulation for a general program running on HPP. Second, we define a particular formulation for stencil computation.

### *4.1.1 Assumptions and simplifications*

In this work, we consider 2D grids since we are essentially targeting image processing and computer vision applications. However, the proposed framework still is applicable for 3D grids. In our formulation, we consider only data transfer between host and device. We omit the representation of host program since we focus essentially on processing on the GPU. A particular emphasis is put on multiple stencil computations running on such platforms. As assumption, we suppose that the studied stencils are symmetric and grids are regular Cartesian grids. The formulation is a single-stencil-based formulation. A single stencil is a stencil that performs a single processing and produces only one output grid, but may consume more than one input grid. The stencils involving more than one processing and more than one output grid need to be decomposed into separate single stencils, so they can be supported by the proposed formulation.

### *4.1.2 Formulation of program running on HPP*

The main formulation terms and their corresponding descriptions are summarized in Table 1.

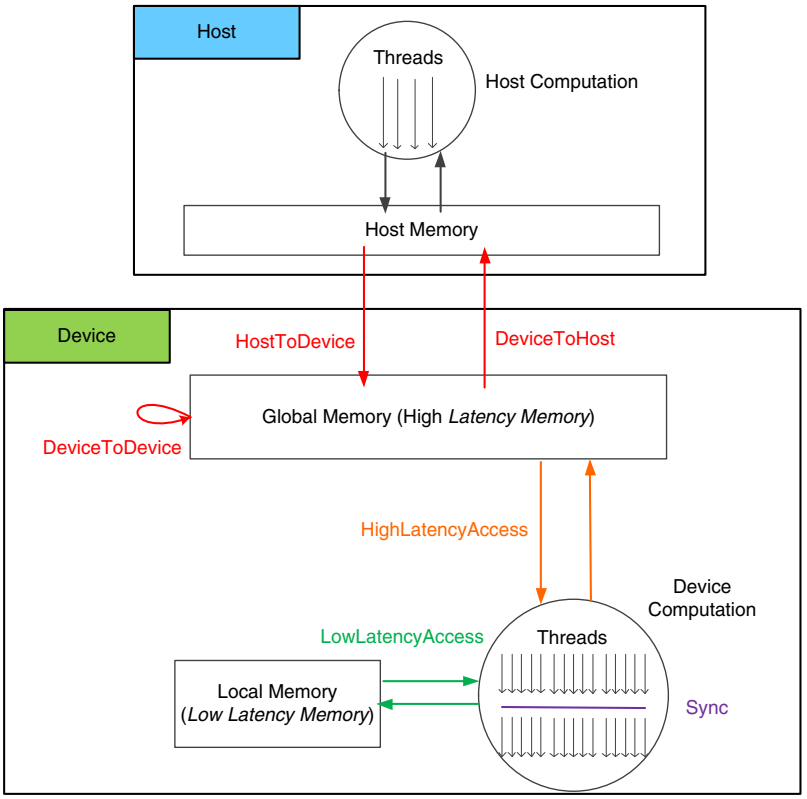


**Fig. 5** Abstract model of heterogeneous parallel platform (HPP)

**Table 1** Summary of main formulation terms and notations

| Symbol | Description |
|---|---|
| $\rightarrow$ | Sequence of actions |
| : | Type |
| $=$ | Definition |
| **Term** | **Description** |
| *DTransfer* | Data transfer between host memory and global memory |
| *DAccess* | Data access from/to global memory or shared memory |
| *Proc* | Processing that includes computation and data access |
| *Comp* | Computation that does not include data access |
| *Sync* | Synchronization mechanism |

At platform-level, we consider two main types of memory *MemType*: (i) high-latency global memory, denoted by $GlobalMem$, and (ii) low-latency scratchpad shared memory, denoted by $ShMem$. Memory type $MemType$ is defined by:

$$MemType: GlobalMem|ShMem. \quad (1)$$

In our formulation, a program running on an HPP is represented as a skeleton expressed as a sequence of processing, denoted by *Proc*, data-transfers, denoted by *DTransfer*, and synchronization, denoted by *Sync*:

$$Program \rightarrow Proc \quad DTransfer \quad Sync. \quad (2)$$

A processing is further expressed as a sequence of data accesses, denoted by $DAccess$, an operation, denoted by $Op$, and synchronization, denoted by $Sync$:

$$Proc \rightarrow DAccess \quad Op \quad Sync. \quad (3)$$

$Proc$ may have different processing types. We focus in particular on stencil processing, detailed in Sect. 4.1.4:

$$ProcType: StencilProc|Other. \quad (4)$$

As for synchronization mechanisms *Sync*, there are of two types:

1. *barrier-based synchronization—BarrierSync*, that denotes the synchronization between threads belonging to the same thread block, and
2. *event-based synchronization—EventSync*, that denotes the synchronization between different streams.

Thus, the definition of the type of synchronization mechanism is defined as follows:

$$SyncType: BarrierSync|EventSync. \quad (5)$$

Still, *DTransfer* can be classified in the following types:

1. $HostToDevice$, which denotes data transfer from the host memory to the device memory;
2. $DeviceToHost$, which denotes data transfer from device memory to host memory, and
3. $Device$, which denotes transfers within the device memory.

Thus, the definition for *DTransferType* is as follows:

$$DTransferType: HostToDevice|DeviceToHost|DeviceToDevice. \quad (6)$$

*Dtransfer* is also defined by the transfer type, denoted by $DtransferType$ and the parameters:

1. $\#ELem$, used to denote the number of transferred elements, and
2. $ElemSize$, used to denote the size (in bytes) of an element, depending on its type ($char$, $integer$, $float$, etc.).

Thus, the definitions of the transfer type are as follows:

$$DTransferDef = DTransferType(\#Elem, ElemSize). \tag{7}$$

A data access type *DAccessType* could be a load or a store operation from/to *GlobalMem* or *ShMem*. Hence, the types of data access: *DAccessType* are defined by:

$$DAccessType: Load|Store. \tag{8}$$

A data access definition $DAccessDef$ is expressed by $DAccessType$ and by (i) the number of accessed elements $\#Elem$, and (ii) the element size $ElemSize$:

$$DAccessDef = DAccessType(MemType, \#Elem, ElemSize). \tag{9}$$

Still, operations can be classified into: (i) integer $I$; (ii) single precision $SP$, and (iii) double precision $DP$. Thus, an operation type $OpType$ is expressed by:

$$OpType: I|SP|DP. \tag{10}$$

Finally, $Op$ is defined by the type of operation $OpType$ and the number of operations: $\#Op$, as follows:

$$OpDef = OpType(\#Op). \tag{11}$$

*4.1.3 Example of program running on HPP*

This section illustrates the presented formulation using a simple image blending application, depicted in Fig. 6, as example. An image is represented as a 2D grid of pixels of size $IM = IM_y \times IM_x$, where $IM_y$ and $IM_x$ are, respectively, the height and the width of the image. Each pixel is located in the grid by its coordinates $y$ and $x$. The blending superposes two input images: $inImg1$ and $inImg2$ that are assigned with different weights by varying a factor $\alpha$. The operation of blending is given by the following equation:

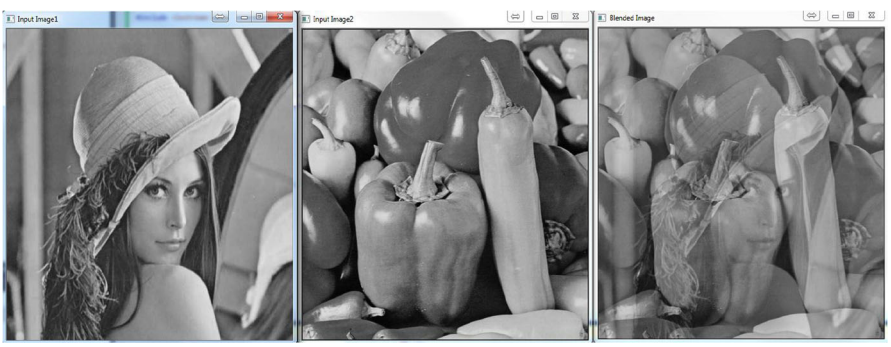


**Fig. 6** Image blending

$$outImg(y,x) = (1-\alpha) \times inImg1(y,x) + \alpha \times inImg2(y,x). \quad (12)$$

The following Listing 1 shows the code skeleton of two grayscale images blending running on HPP, expressed using the proposed formulation terms. The input images of size $IM_y \times IM_x$ are transferred from $HostMem$ to $GlobalMem$. The images stored in $GlobalMem$ are divided into tiles of size $(TL = TL_y \times TL_x)$ over the parallel thread blocks. Each thread block $Blk$ iterates over its assigned number of tiles denoted by $\#TL$. At the last stage, the resultant tile is stored in the $GlobalMem$.

**Listing 1** Code skeleton of blending program

```
DTransfer:HostToDevice(IM, 1);
DTransfer:HostToDevice(IM, 1);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, TL, 1);
    DAccess:Load(GlobalMem, TL, 1);
    Op:I(TL);
    DAccess:Store(GlobalMem, TL, 1);
}
DTransfer:DeviceToHost(IM, 1);
```

#### *4.1.4 Formulation of stencil-based processing*

After defining a program running on HPP, this section shows a particular type of processing—*StencilProc*, which is the stencil-based processing. *StencilProc* is defined by the following sequence:

$$StencilProc \rightarrow DAccess(Stencil) \quad Op(Stencil) \quad Sync(Stencil). \quad (13)$$

In our framework, we define as simple stencil the one that consumes one or multiple grids but produces only one grid. In typical stencil processing, each input grid is divided into tiles of size $TL$ over a number of thread blocks $\#Blk$. In order to accelerate processing via accessing low-latency memory $ShMem$, each tile is loaded to $ShMem$ to be processed by a given thread block.

Since each thread block has a view of only a part of the data, restricted to what is loaded on the local address space allocated for it, the problem of dependency at borders will arise. To handle it efficiently, a well-known technique called *overlapped tiling* [12] is employed. This technique consists in augmenting each tile with extra halo elements $H$, as shown in Fig. 7.

In this case, we define each tile size $(TL = TL_y \times TL_x)$ and each halo size $(H = H_y \times H_x)$, so the augmented tile of size $TL'$ loaded into the shared memory is defined by:

$$TL' = (TL_y + 2 \times H_y) \times (TL_x + 2 \times H_x). \quad (14)$$

A stencil is defined in our formulation by its identifier $StencilID$ and the parameters:

1. a set of pairs of input grid $inGrid_i$ and its corresponding halo $H_i$;

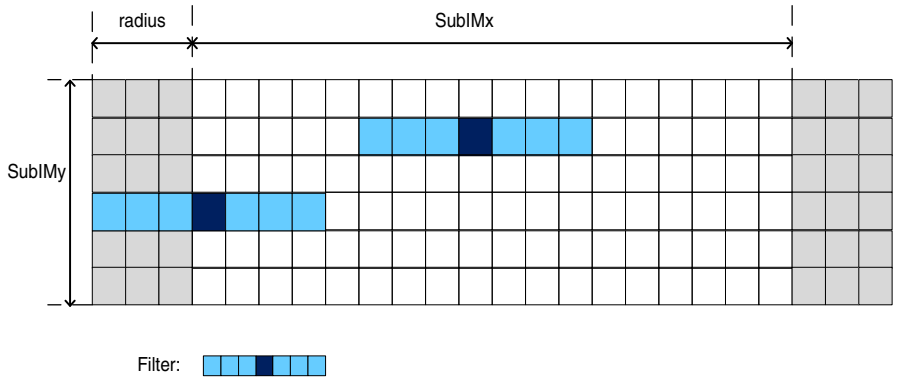


**Fig. 7** Convolution on rows

2. an output grid $outGrid$, and
3. a transition function *TransFunc* shown in expression (15).

In stencil computation, each element of the output grid is calculated depending on the element's values at its current position and its neighborhood of an input grid, by applying a transition function $TransFunc$. For each dimension $dim$, the size of the stencil is calculated as $(2 \times H_{dim} + 1)$, where $H$ is the halo added to the input grid to compute the elements of the output grid.

$$StencilDef = StencilID\left(\bigcup_{i}^{\#inGrid} \{(inGrid_i, H_i)\}, outGrid, TransFunc\right) \tag{15}$$

$Grid$ defines an $N$-dimensional Cartesian grid when has its identifier $GridID$, its number of dimensions $\#Dim$, its size $Size = \prod_{dim}^{\#Dim} Size_{dim}$ and the size of each element $ElemSize$, expressed in bytes.

$$GridDef = GridID(\#Dim,\ Size,\ ElemSize) \tag{16}$$

*TransFunc* defines the stencil operation $Op$ on one grid element and is expressed by:

$$TransFuncDef = OpType(\#Op). \tag{17}$$

The number of operations, denoted by $\#Op$ is given by the following expression:

$$\#Op = \sum_{i}^{\#inGrids} \left(\prod_{dim}^{\#Dim} ((2 \times H_{i,dim} + 1)\right) \tag{18}$$

*4.1.5 Example of stencil-based processing*

As example of stencil-based processing, we consider a simple image convolution on rows. The input image is of size $IM = IM_y \times IM_x$, where $IM_y$ and $IM_x$ are,

respectively, the height and the width of the image. The operation of convolution is performed on each pixel of the image by applying an 1D filter (of aperture size of $2 \times radius + 1$) centered on that pixel. The result of convolution is stored at the centered pixel in the output image as shown previously in Fig. 7. The convolution operation is given by:

$$outImg(y, x) = \sum_{-radius}^{radius} inImg(y, x + k) \times filter(k + radius). \tag{19}$$

The convolution processing of the defined formulation for stencil computation is given in expression (20). $Conv$ is the transition function that performs the convolution operation and it is defined by expression (21).

$$Stencil(Convolution) = StencilConv((inImg, radius), outImg, Conv) \tag{20}$$

$$Conv = SP(\#Op(Conv)) \tag{21}$$

$$\#Op(Conv) = 2 \times radius + 1. \tag{22}$$

In the following, we present the code skeletons of two different versions:

1. *the first version*, without shared memory usage (see Listing 2 for details), and
2. *the second version*, which uses the shared memory (see Listing 3 for details).

For both versions, the input image of size $IM$ is transferred from $HostMem$ to $GlobalMem$. The image stored in $GlobalMem$ is divided into tiles of size ($TL = TL_y \times TL_x$) over the parallel thread blocks. Each thread block $Blk$ iterates over its assigned number of tiles denoted by $\#TL_{Blk}$. For convolution that uses shared memory, each tile is augmented with an extra halo: ($2 \times radius$) and stored in $ShMem$. The size of the augmented tile is denoted by $TL'$ is given by:

$$TL' = TL_y \times (TL_x + 2 \times radius). \tag{23}$$

Also, in both versions, the convolution is applied on each tile element by performing ($2 \times radius + 1$) loads and single precision operations. The output tile is stored to $GlobalMem$ and all the output tiles forming the output image $outImg$ are transferred back to the host memory.

**Listing 2** Code skeleton of convolution without shared memory

```
W = 2*radius+1;
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, W × TL, 4);
    Op:I(W × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

**Listing 3** Code skeleton of convolution with shared memory

```
W = 2*radius+1;
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    DAccess:Load(GlobalMem, TL', 4);
    DAccess:Store(ShMem, TL', 4);
    Sync:BarrierSync
    DAccess:Load(ShMem, W × TL, 4);
    Op:SP(W × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

## 4.2 Fusion and thread configuration (step 2)

In this section, we define the second step of our methodology. The program characteristics, such as the data grid size, its dimensions, and the halo size are the main input. This step is divided in two sub-steps:

1. *sub-step 1* for each tile placed in the shared memory, it computes the total halo size needed to be added to avoid any extra data exchange with the global memory. This step is performed for each fusion combination defined by the developer and depends on the halo size defined for each stencil during the stencil formulation at Step 1. At the end of this sub-step, the total halo size for each tile is extracted and injected to the second sub-step, and
2. *sub-step 2* it takes as input both the number of tiles placed in shared memory and the halo size computed previously. It generates a number of possible configuration under the architecture constraints. For each generated configuration, a set of resource usage is also provided.

### *4.2.1 Stencil fusion problem*

In iterative stencil computation, *stencil fusion* (also called *time tiling*) is an optimization technique, which consists of grouping a number of stencils together. In our case, this optimization is done in a single step, where all possible configurations are generated automatically. This technique allows better data locality, especially in the case of GPUs. Since the data access to global memory has an expensive computational cost, the idea is to fuse several stencils and let the intermediate data resident in the shared memory for as long as possible. However, the fusion has some side effects, such as extra data load and re-computation overheads, which are necessary to avoid data updates via the access to global memory.

The implementation of fusion on GPUs is not an easy task especially when dealing with multiple stencil shapes and data grids. The developer has the challenge of determining the size and the shape of halos that will be added to each involved grid in the computation of the fused stencils. Also, another challenge is to figure out which stencils should be fused. An example of different stencil combinations is shown in Fig. 8.

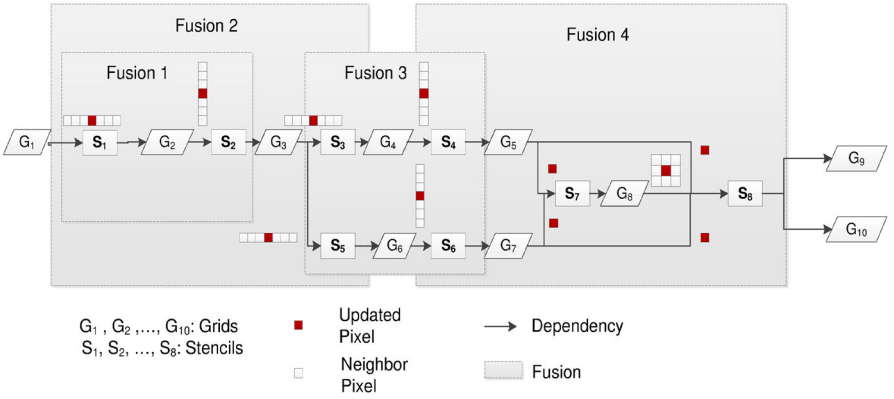


**Fig. 8** Fusion combinations

In order to help developers to implement the best stencil fusion combination, we provide a tool to compute the appropriate halo size and shape to be added to each involved data grid for a selected fused stencils. The algorithm implemented at this stage can be seen in Algorithm 1 and is described below. The output of this tool serves as input for a second tool, which provides all possible thread mapping configurations and their corresponding GPU resource usage for the selected fused stencils. The second tool is described in Sect. 4.2.3. At the final stage, an analytical performance model (described in Sect. 4.5) is used to provide the possible configurations according to their computational costs.

**Algorithm 1:** Halo size computation algorithm

**Input** : Weighted DAG of Fused Stencils
**Output**: $H_i'$: Total Halo Size for each $shGrid_i$
**foreach** $shGrid_i$ **do**
  Intialize $H_i' = 0$
**end**
**foreach** $outGrid_j$ **do**
  **foreach** $shGrid_i$ **do**
    Compute $H'(i, j) = \text{LongestPath}(shGrid_i, outGrid_j)$
    Update $H_i' = max(H_i', H'(i, j))$
  **end**
**end**

A stencil-based application shown as input application in Fig. 9 could be represented as a weighted direct acyclic graph (DAG).

In this case, vertices represent the data grids $G_i$, the edges represent the stencils and their weights represent the halo size $H$ to be added for the stencil computation. Each halo is denoted by $H(inGrid, outGrid)$, which defines the halo to be added to $inGrid$ to compute $outGrid$. When we apply fusion to a group of stencils, all grids are loaded in the shared memory with the exception of the output grids, represented

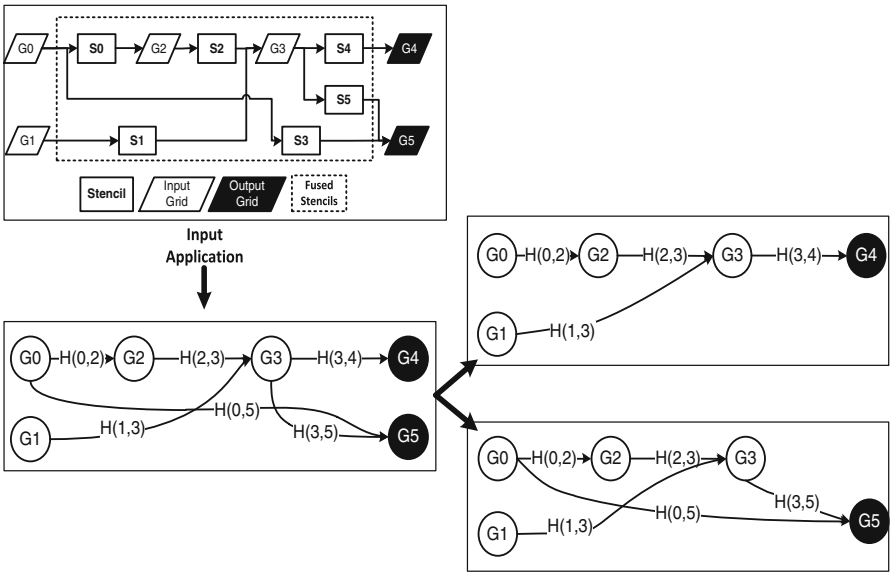


**Fig. 9** Representation of fused stencils

as black vertices. All grids loaded in the shared memory denoted by $shGrids$ are represented as white vertices.

The total halo size to be added to grids loaded in shared memory to update the output grids needs to be determined in order to solve the dependency problem and to avoid any extra data exchange with the global memory. For this purpose, the initial graph is treated as two separate graphs, being one for each output grid.

Then, for each graph we determine the halo size to be added to each $shGrid$ in order to compute the output grid for that graph. A halo with size $H'(i, j)$ is added to $shGrid_i$ in order to compute $outGrid_j$. To find this size, the problem is reduced to a longest path problem. In this case, $H'(i, j)$ is the longest path length separating the vertices $shGrid_i$ and $outGrid_j$. At the final stage, the total halo size is the maximum size among all the halo sizes to be added to compute each output grid, as shown in Eq. 24.

$$H'_i = \max_j(H'(i, j)), \quad \forall i \in shGrids, \quad \forall j \in outGrids. \tag{24}$$

### *4.2.2 Example of stencil fusion*

To illustrate our approach, we apply fusion on an image processing application. As example we use the Gaussian blur filter operation, which consists of two successive separate 1D convolution operations applied on rows and on columns. Gaussian blur can be represented as two successive stencils, as shown in Fig. 10. The first stencil is a 1D horizontal convolution involving a halo of size $r$. The second stencil is a 1D vertical convolution involving a halo of size $r$. The representation of the two stencils using our formulation is shown in Table 2. The code skeleton of the fused Gaussian blur is illustrated in Listing 4. In this case, the halos added to each grid $G_1$ and $G_2$, in the shared memory, are given by:

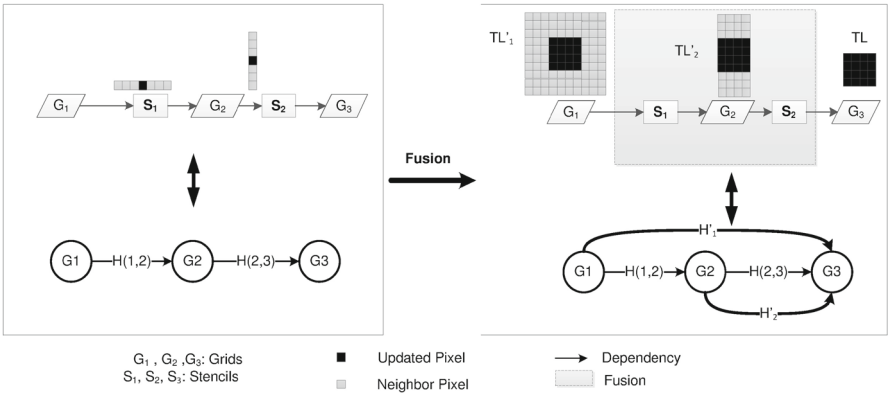


**Fig. 10** Gaussian blur fusion

**Table 2** Gaussian blur stencils

| Operation | Stencil | $\bigcup_i^{\#inGrid}\{inGrid_i : (H_{i,y}, H_{i,x})\}, outGrid$ |
|---|---|---|
| Convolution on row | $S1$ | $G1: (0, r), G2$ |
| Convolution on col. | $S2$ | $G2: (r, 0), G3$ |

$$\begin{cases} H'_2 = H(2,3) = r \times 1 \\ H'_1 = H(1,2) + H(2,3) = r \times r. \end{cases} \tag{25}$$

**Listing 4** Code skeleton of fused Gaussian blur

```
H'_1 = (r, r);
H'_2 = (r, 0);
TL'_1 = (TL_y + H'_{1,y}) × (TL_x + H'_{1,x});
TL'_2 = (TL_y + H'_{2,y}) × (TL_x + H'_{2,x});
DTransfer:HostToDevice(IM, 4);
For i = 1 To #Blk_SM
{
    \\load Input Tile in ShMem
    DAccess:Load(GlobalMem, TL'_1, 4);
    DAccess:Store(ShMem, TL'_1, 4);
    Sync:BarrierSync
    \\S1
    DAccess:Load(ShMem, (2 × H_1 + 1) × TL'_2, 4);
    Op:SP((2 × H_1 + 1) × TL'_2);
    DAccess:Store(ShMem, TL'_2, 4);
    Sync:BarrierSync
    \\S2
    DAccess:Load(ShMem, (2 × H_2 + 1) × TL, 4);
    Op:SP((2 × H_1 + 1) × TL);
    DAccess:Store(GlobalMem, TL, 4);
}
DTransfer:DeviceToHost(IM, 4);
```

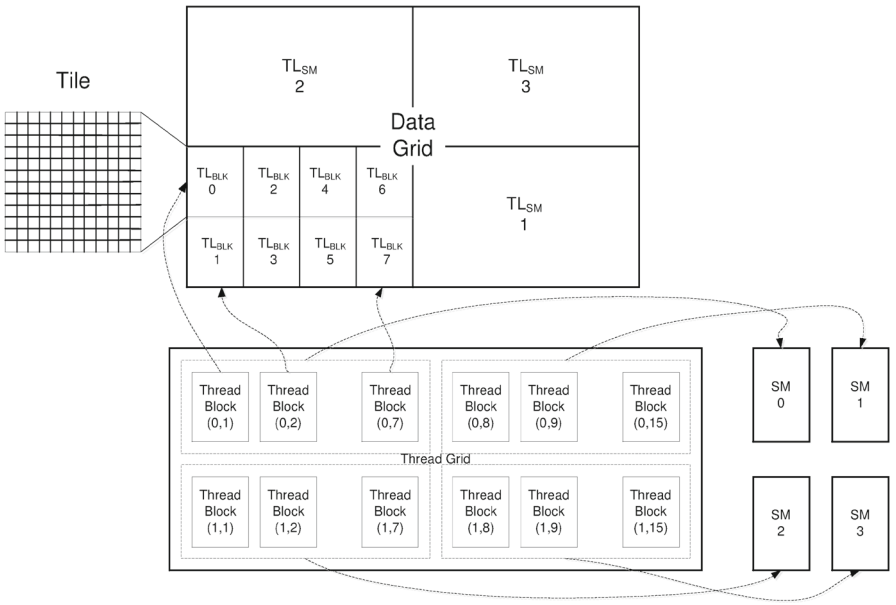


**Fig. 11** Tiling and thread block configuration

### *4.2.3 Tile size and thread block configuration problem*

Tile size and thread mapping problem is depicted in Fig. 11 and can be reduced to these questions:

- How to divide the data grid into tiles?
- How to assign thread blocks to tiles?
- How thread blocks may be distributed across streaming multiprocessors?

It is important to do a good choice of the thread block configuration and the tile size processed by each block. To highlight that, we show an example of performance differences of different configurations of the convolution program on three GPU architectures: (i) GTX 590 (Fermi), (ii) GTX 780 (Kepler) and (iii) GTX 750 (Maxwell).

Each configuration is a tuple of parameters $(T_y, T_x, N_y, N_x)$, where the first two define the thread block size, and the last two define the number of elements processed by each thread. The tile size processed by each thread block is given by Eq. 26. Figure 12 shows the execution time of several configurations. We observe that performance vary significantly from a range of configurations to another. Also, we observe that the best configurations set varies from one architecture to another.

$$TL = (T_y \times N_y) \times (T_x \times N_x). \tag{26}$$

To deal with the large space of possible configurations and to guide the choice of the best set of configuration, we developed a tool that takes as input:

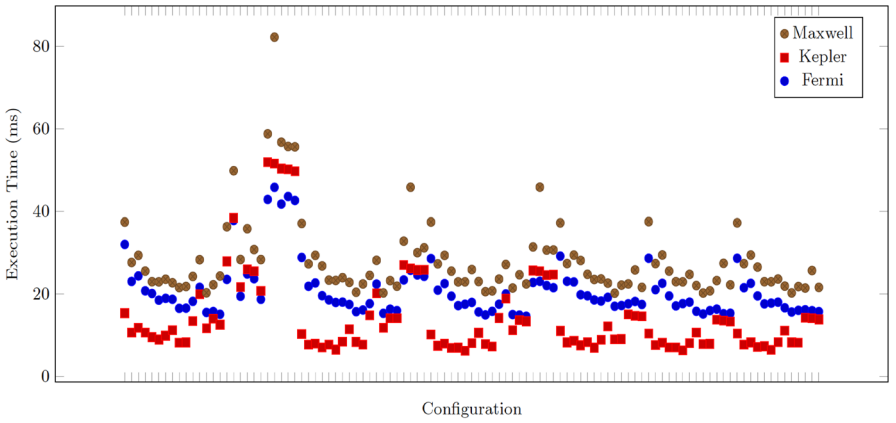


**Fig. 12** Execution time of convolution on rows kernel (ms)

**Table 3** Platform specification

| Parameter | Description | GTX 590 (2.0) | GTX 780 (3.5) | GTX 750 (5.0) |
|---|---|---|---|---|
| $T_{Blk,dim}^{Max}$ | Maximum # of threads per block at the dimension $dim$ | 1024 | 1024 | 1024 |
| $T_{Blk}^{Max}$ | Maximum # of threads per block | 1024 | 1024 | 1024 |
| $T_{SM}^{Max}$ | Maximum # of threads per SM | 1536 | 2048 | 2048 |
| $Blk_{SM}^{Max}$ | Maximum # of blocks per SM | 8 | 16 | 32 |
| $Smem_{Blk}^{Max}$ | Maximum ShMem per block | 48 KB | 48 KB | 48 KB |
| $Smem_{SM}^{Max}$ | Maximum ShMem per SM | 48 KB | 48 KB | 64 KB |
| $\#SM$ | Number of SMs | 16 | 12 | 4 |

1. *the halo size* involved for each stencil loaded in the shared memory. These halo sizes are computed with the first stage of the tool described in Sect. 4.2.1, and
2. *the GPU architecture specification* as well as the resource constraints, which are defined as follows and summarized in Table 3.

This tool offers an exhaustive coverage of all possible configurations that may be implemented for a given application on a given GPU architecture. The algorithm responsible for computing the resource usage for each possible configuration is described in Algorithm 2.

For each configuration, the initial tile size is set by Eq. 26. The number of elements processed by each thread at dimension $y$ and $x$ are denoted as $N_y$ and $N_x$, respectively. A thread block is defined by $T_y$ and $T_x$ to denote the $\#Threads$ at each dimension. To augment the initial tile by halo, we use the padding technique, which consists on adding extra elements to the halo in order to align the augmented tile size to the thread block size. This technique helps to avoid divergent computation paths and allows better alignment with memory access. Therefore, both $T_y$ and $T_x$ must be greater than the maximum halo size on, respectively, $y$ and $x$ dimension. Equation 27 defines the size of the augmented tile with the needed halo for the stencil computation. The halo added to the initial tile is given by $H'_y$ and $H'_x$ for, respectively, $y$ and $x$ dimension. The used shared memory per block is defined in Eq. 28, where $Smem_{Blk}$ is the total size of the augmented tiles for each input grid. The $\#ResBlk_{SM}$ expressed in Eq. 29 is determined with respect of the platform constraints and $Smem_{Blk}$.

$$TL'_i = \left(N_y + \left\lceil \frac{H'_{i,y}}{T_y} \right\rceil\right) \times \left(N_x + \left\lceil \frac{H'_{i,x}}{T_x} \right\rceil\right) \times (T_y \times T_x) \backslash \forall inGrid_i \quad (27)$$

$$Smem_{Blk} = \sum_{i}^{\#inGrids} (TL'_i \times ElemSize(inGrid_i)) \quad (28)$$

$$\#ResBlk_{SM} = \min\left(\left\lfloor \frac{Smem^{max}_{SM}}{Smem_{Blk}} \right\rfloor, \left\lfloor \frac{T^{max}_{SM}}{T_{Blk}} \right\rfloor, \#Blk^{max}_{SM}\right) \quad (29)$$

**Algorithm 2:** Resource usage computation algorithm

**Input** : $G_i : (H_{i,y}, H_{i,x})$: Halo size needed for the number of tiles in $ShMems$
**Input** : $(IM_y, IM_x)$: Data grid size
**Input** : $\#SM, Smem^{Max}_{Blk}, Smem^{Max}_{SM}, Blk^{Max}_{SM}, T^{Max}_{Blk}, T^{Max}_{SM}$: Architecture specification
**foreach** *configuration* $(Th_y, Th_x, N_y, N_x)$ **do**
  **if** $T_y \geq 2 \times H_y$ ***and*** $T_x \geq 2 \times H_x$ ***and*** $T_y \times T_x \leq Th_{Blk}$ **then**
    Compute $Smem_{Blk} = \sum_i TL'_i \times ElemSize(inGrid_i)$;
    Compute $\#ResBlk = Min(\lfloor \frac{Smem_{SM}}{Smem_{Blk}} \rfloor, \#Blk^{Max}_{SM})$;
    Compute $Occupancy = \frac{\#ResBlk \times T_y \times T_x}{T^{Max}_{SM}}$;
    Compute $\#iter_{SM} = \lfloor \frac{IM}{\#SM \times \#ResBlk_{SM} \times TL} \rfloor$;
    Compute $\#RemBlk_{SM} = \lceil \frac{IM - \#iter \times \#ResBlk_{SM} \times \#SM \times TL}{T_{Blk}} \rceil$;
  **end**
**end**

### 4.3 Influential factors on performance (step 3)

This section defines the third step of our methodology, where we provide a list of the main influential factors on GPU performance. These factors are used to setup the performance model. This step is performed only if there is a new used architecture

where the impact of the influential parameters on performance vary. In the following, we present the list of the main influential factors on performance:

- *halo size*, as each data grid is divided into tiles where each tile is augmented with extra neighbor elements (halo) to respect dependencies at borders. The halo size depends on the way the grid is split and on the properties of computations as depicted previously in Fig. 3. If we divide the grid according to the orthogonal direction to that where the larger number of neighbors are involved, the size of halo increases and yields additional data load and recomputation overheads;
- *thread occupancy*, which is the rate of actual resident threads by the maximum allowed resident threads per SM (see Eq. 30). Low thread occupancy prevents both memory latency and arithmetic latency hiding;
- *branch divergence*. When threads from the same warp follow different paths due to conditional statements, their execution is serialized.

$$Occupancy = \frac{\#ResBlocks}{Blk_{SM}^{Max}} \tag{30}$$

- *memory access coalescing and alignment*. Aligned memory accesses occur when the first address of a device memory is an even multiple of the cache granularity used to service the transaction (either 32 bytes for L2 cache or 128 bytes for L1 cache). Coalesced memory accesses occur when all 32 threads in a warp access a contiguous chunk of memory. When threads access non-coalesced memory locations we see the performance dropping significantly, and
- *shared memory bank access*. If different parallel threads from the same warp access the same memory bank, the memory access to shared memory is serialized.

### 4.4 Controlled parameters

In this section, we provide the parameters related to influential factors on performance and that can be controlled by the developer at the programming level. These parameters are:

1. *Data grid decomposition and fusion.* In order to minimize the halo size and increase the effective computation rate, the developer has to manage the data grid decomposition according to the tile size and shape. Another parameter that has an immediate impact on both halo size and thread occupancy is the total size of fused grids stored in the shared memory. The developer has to select the number of stencils and which ones to fuse, in order to reach high-performance computation.
2. *Thread block configuration.* HPP runtime for both CUDA and OpenCL organize threads into grids of thread blocks and these blocks may have 1D, 2D or 3D layout. The developer can choose the thread block size and its shape. However, this choice depends on several constraints imposed by both hardware architecture and the runtime constraints, as mentioned in Sect. 4.2.3.
   - *Thread block size*—it affects the occupancy rate and by consequence: (i) the degree of concurrency, and (ii) both arithmetic operations and memory latency

hiding. Thread block size has also an impact on the amount of the remaining work load. By selecting a small block size, many blocks could be mapped into one SM. This allows better fit with shared memory size when the problem size is not regular and does not match exactly the resource limits. Also, since the number of blocks is not distributed equally to the available SMs, small block size gives better load balancing. However, small block size introduces more overhead due to the number of halos that have to be loaded. On the other side, by selecting a large block size, fewer blocks could be mapped into the SM in order to respect the maximum number of threads allowed per SM. In many cases, this will introduce a bad fit with the resource constraints and, by consequence, an inefficient use of available computation capability of the hardware platform. However, since the number of loaded halos is proportional to the number of thread blocks, the overhead due to the halo loading and recomputation is minimized.

- *Thread block layout*—it affects the memory access efficiency. The number of threads $T_x$ has an immediate impact on the memory alignment and banked shared memory access efficiency. Examples of memory efficiency impact of $T_x$ for 1D horizontal stencil, 1D vertical stencil and 2D stencil are shown, respectively, in Tables 4, 5, and 6. The number of transactions per memory request for both shared memory and global memory is immediately related to $T_x$. We observe that the number of transaction per request to shared memory doubles when $T_x$ is not multiple of a memory banks number size in the 1D horizontal

**Table 4** # Transactions per request (1D horizontal stencil)

| $T_x$ | Shared load | Shared store | Global load | Global store |
|---|---|---|---|---|
| 8 | 2 | 2 | 5.72 | 8 |
| 16 | 2 | 2 | 4.83 | 4 |
| 32 | 1 | 1 | 4.35 | 4.42 |
| 64 | 1 | 1 | 4 | 4 |
| 128 | 1 | 1 | 4.29 | 4 |

**Table 5** # Transactions per request (1D vertical stencil)

| $T_x$ | Shared load | Shared store | Global load | Global store |
|---|---|---|---|---|
| 4 | 1 | 1 | 7.7 | 8 |
| 8 | 1 | 1 | 4 | 8 |
| 16 | 1 | 1 | 4 | 4 |
| 32 | 1 | 1 | 4 | 4 |
| 64 | 1 | 1 | 4 | 4 |

**Table 6** # Transactions per request (2D stencil)

| $T_x$ | Shared load | Shared store | Global load | Global store |
|---|---|---|---|---|
| 8 | 1 | 1 | 4.79 | 8 |
| 16 | 1 | 1 | 4.34 | 4 |
| 32 | 1 | 1 | 4.16 | 4 |
| 64 | 1 | 1 | 4.51 | 4 |

stencil computation. The number of transactions per request to global memory doubles when $T_x$ is not multiple of a half warp, and this number increases as $T_x$ decreases. The thread block layout has also an impact on the halo loading and recomputation overhead. Depending on the halo layout imposed by the application, the user has to choose the better block layout that minimizes this overhead and utilizes better the hardware computation capability.

### 4.5 Performance model for stencil computation (step 4)

This section defines the fourth and final step of our methodology. At this level, we present our performance model used to compute the time cost of stencil-based processing on NVIDIA GPUs. This performance model takes two inputs:

1. *the program characteristics* defined at the first step as well as the resource usage and the computation amount, and
2. *the influential parameters on performance* that were defined at the third step and are used to set up our performance model.

We provide a general performance model that considers the computation, the synchronization and the memory operations as basic processing operations as those mentioned previously. The proposed performance model focuses on device processing times.

The processing time ($T_{proc}$) is defined as a sum of: (i) the data access time ($T_{DAccess}$); (ii) the synchronization time ($T_{Sync}$), and (iii) the operation time ($T_{Op}$). Equation 31 brings the formal definition.

$$T_{Proc} = T_{DAccess} + T_{Sync} + T_{Op}. \tag{31}$$

The operation time is expressed in Eq. 32 as a function of (i) the operation latency denoted by $Lat_{Op}$; (ii) the number of performed operations denoted by $\#Op$, and (iii) the operation throughput denoted by $OpTh$.

$$T_{Op} = Lat_{Op} + \frac{\#Op}{OpTh(OpType)}. \tag{32}$$

Table 7 gives the throughputs in number of operations per clock cycle of the main operations for different NVIDIA GPU architectures.

The operation latency could be expressed as a function of the number of resident warps $\#ResWarp$ per SM (see Eq. 33).

$$\begin{cases} Lat_{Op} = f(\#ResWarp) \\ Lat_{DAccessType} = f(\#ResWarp) \\ \#Trans_{DAccessType} = f(T_x) \end{cases} \tag{33}$$

$Lat_{Op}$ is equal to zero if $\#ResWarp$ is greater than a minimum number of resident warps denoted by $\#ResWarp_{min}$. Based on the NVIDIA programming guide,

**Table 7** Throughput of main operations per clock cycle [7]

| Compute capability | 2.0 | 3.5 | 5.0 |
|---|---|---|---|
| 32-Bit floating-point add, multiply, multiply-add | 32 | 192 | 128 |
| 64-Bit floating-point add, multiply, multiply-add | 16 | 64 | 1 |
| 32-Bit square root, special functions | 4 | 32 | 32 |
| 32-Bit integer add, subtract | 32 | 160 | 128 |
| 32-Bit integer multiply, multiply-add | 16 | 32 | 32 |

**Table 8** Average memory latency in clock cycles [7]

| Compute capability | 2.0 | 3.5 | 5.0 |
|---|---|---|---|
| Global memory latency (clock cycle) | 600 | 300 | 300 |
| Shared memory latency (clock cycle) | 16 | 16 | 16 |
| Global memory clock rate (MHz) | 4008 | 6008 | 5000 |
| Global memory bus width (bit) | 384 | 384 | 256 |
| Theoretical global mem. bandwidth (GB/s) | 164 | 288 | 224 |
| Shared mem. bandwidth (GB/s) | 1476 | 2592 | 2016 |

$ResWarp$ has to be at least 24 warps for Fermi and 44 warps for Kepler to hide latency operations. Data access time denoted by $T_{DAccess}$ is expressed in Eq. 34 as a sum of latency time denoted by $Lat_{DaccessType}$ and the time needed to access $\#Elem \times ElemSize$ bytes.

$$T^{MemType}_{DAccessType} = Lat^{MemType}_{DAccessType} + \frac{\#Trans^{MemType}_{DAccessType} \times \#Elem \times ElemSize}{BW_{MemType} \times Warp} \tag{34}$$

Table 8 gives the latency in clock cycle and the bandwidth of both shared memory and global memory for different NVIDIA GPU architectures.

The total processing time on device is given by the processing time of the most loaded SM. First, we express the maximum work load that could be assigned to one SM by using the model given below. The number of thread block assigned to an SM is denoted by $\#Blk_{SM}$. It is expressed in Eq. 35 as a number of compute iterations denoted by $\#iter_{SM}$ multiplied by the number of resident blocks per $SM$ denoted by $\#ResBlk_{SM}$ and the number of remaining blocks denoted by $\#RemBlk_{SM}$ and expressed in Eq. 36.

$$\#Blk_{SM} = \#iter_{SM} \times \#ResBlk_{SM} + \#RemBlk_{SM} \tag{35}$$

$$\#RemBlk_{SM} = \left\lceil \frac{IM - \#iter \times \#ResBlk_{SM} \times \#SM \times TL}{T_{Blk}} \right\rceil \tag{36}$$

The $\#iter_{SM}$ is expressed in Eq. 37 as the number of iterations needed per $SM$ to compute its portion of the total data grid of size $IM$ at the rate of $\#ResBlk_{SM}$ per computation step, where each resident thread block computes one tile of size $TL$.

$$\#iter_{SM} = \left\lfloor \frac{IM}{\#SM \times \#ResBlk_{SM} \times TL} \right\rfloor \quad (37)$$

To ensure the load balancing, our framework employs a number of techniques such as the overlapped tiling technique and the padding technique. The work proposed in [11] shows that the overlapped tiling technique in the context of multi-core CPUs reduces the communication costs and improves load balancing at the price of relatively low overhead introduced by the replicated computation. The padding technique eliminates divergent paths which ensures homogeneous workload for all threads. In addition, our framework provides a metric that is related directly to the load balancing which is the number of remaining blocks per SM. This metric provides the extra work allocated to some SMs. Since this extra work affects the total runtime of a projected configuration, this value is minimized for the kept optimal configurations and accordingly the load balancing is more respected. To validate the load balancing aspect, we perform a profiling of the optimal configurations using the NVIDIA Nsight tool.

## 5 Experimental results

In order to validate our framework, we implement two real image processing applications on three NVIDIA GPU architectures using the developed tuning tool. First, we provide the experimental results of a relatively simple application. Second, we provide the experimental results of a more complex application involving multiple stencils.

### 5.1 Experimental environment

We use two main hardware platforms as host:

1. 2x AMD Opteron 6128 CPU with total of 16 cores running at 2 GHz, and
2. AMD 5800K integrating four cores running at 4.3 GHz.

In the first platform, our experiments are performed on a Maxwell NVIDIA GTX 750 GPU. In the second platform, our experiments are performed on two NVIDIA architectures: Fermi with NVIDIA GTX 590 and Kepler with NVIDIA GTX 780. As programming model, we use CUDA to program the NVIDIA GPUs.

### 5.2 Case study: Gaussian blur

Gaussian blur is represented as two successive stencils. First, we compare the performance of the configuration obtained by the tuning tool with an SDK implementation for two different NVIDIA GPU architectures. The tuned configuration is denoted as *Tuned S1* and *Tuned S2* for the stencils $S1$ and $S2$, respectively. *SDK S1* and *SDK S2* denote the stencils implemented in the CUDA SDK with a *default* thread block configuration. The experiments are performed on two image sizes: (1) $4096 \times 4096$ and (2) $8192 \times 8192$. The target GPUs are GTX 590, GTX 780 and GTX 750. All the details about the implemented stencils are given in Table 9.

**Table 9** Gaussian blur stencils

| Operation | Stencil | $\bigcup_i^{\#inGrid}\{inGrid_i : (H_{i,y}, H_{i,x})\}, outGrid$ |
|---|---|---|
| Convolution on row | $S1$ | $G1: (1, 7), G2$ |
| Convolution on col. | $S2$ | $G2: (7, 1), G3$ |
| Fused Gauss | $(S1, S2)$ | $G1: (1, 7), G2: (7, 1), G3$ |

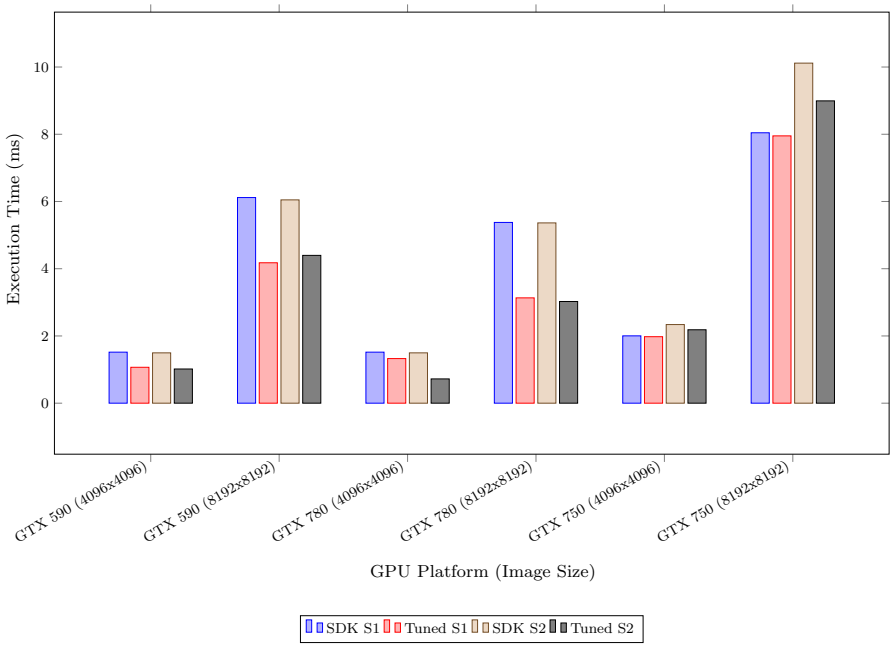


**Fig. 13** Performance comparison of different separate Gaussian blur stencil implementations

We show that the *tuned* configuration outperforms the default SDK implementation in all cases. The gain of the tuned implementation over the default SDK implementation varies depending on the architecture and the image size. The gain is more relevant in the case of large images and in particular for GTX 780 compared to little gain for GTX 750. All the experimental results are shown in Fig. 13.

Next, Fig. 14 depicts the experimental results of an implementation of both non-fused Gaussian and fused Gaussian blur on three NVIDIA GPU architectures. We observe that the tuned configuration outperforms a basic implementation having the configuration $(T_y, T_x, N_y, N_x) = (16, 16, 1, 1)$ for both non-fused and fused versions.

### 5.3 Solution space exploration

The tuned implementation of a stencil computation for a particular GPU architecture is reached by exploring the large solution space which includes all possible configu-

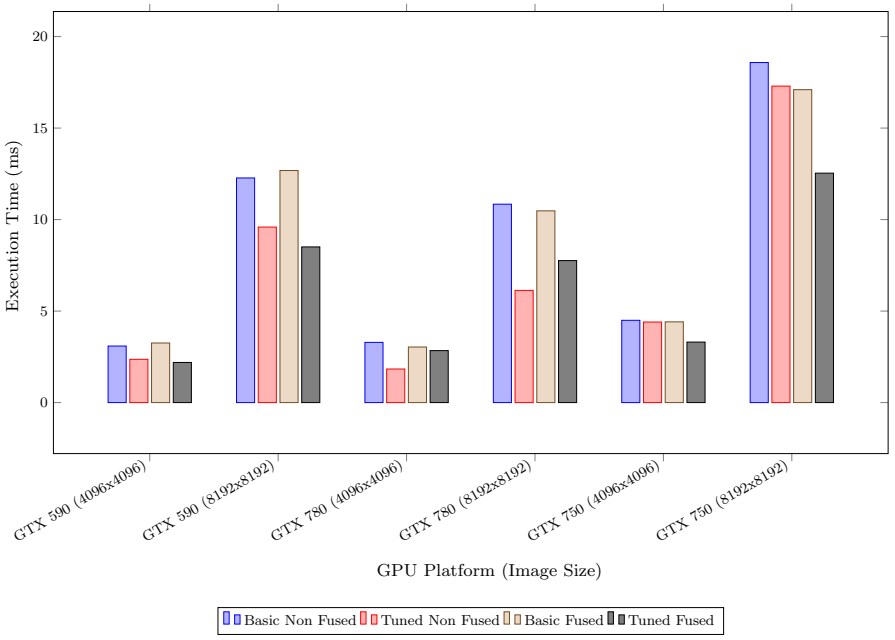


**Fig. 14** Performance comparison of different implementations of non-fused and fused Gaussian blur

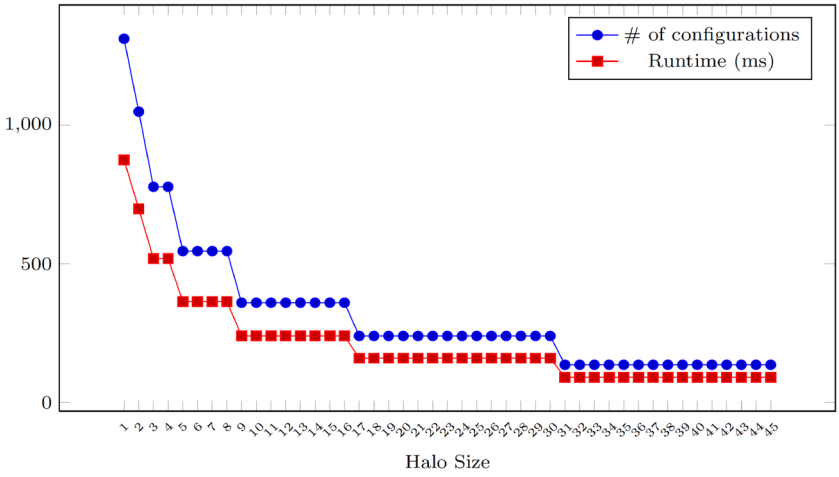


**Fig. 15** Exploration runtime for convolution kernel with different filter sizes

rations. In the proposed framework, we perform an efficient exploration space based on a number of guidelines provided by the GPU vendors according to the thread occupancy ratio and the thread block sizes. This optimization enables to reduce the exploration space size and the exploration runtime. In Fig. 15, we represent the results of an exhaustive search for a convolution on rows with different filter sizes. Both the number of configurations and the corresponding exploration runtime are repre-

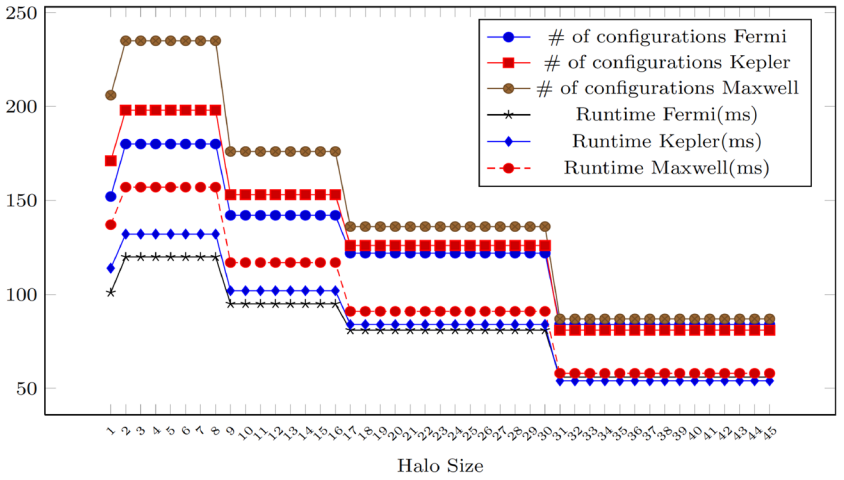


**Fig. 16** Filtered exploration runtime for convolution kernel with different filter sizes

sented. The optimized exploration is represented in Fig. 16. The exploration runtime is reduced considerably since only relevant configurations for each GPU architecture are explored.

### 5.4 Case study: Canny edge detection application

To evaluate our tuning tool, we study the Canny edge detection (CED) algorithm, which can be expressed as a sequence of several stencil operations. A representation of CED under the form of a sequence of stencils is illustrated in Fig. 17.

The stencil $S1$ is a convolution on the horizontal direction and the stencil $S2$ is the convolution in the vertical direction. The sequence of these two stencils form the Gaussian blur stage. The sequence consisting in $S3$ and $S4$, and the sequence consisting in $S5$ and $S6$ form, respectively, the gradients at $X$ and $Y$ direction stages. Both $S3$ and $S5$ are convolution in the horizontal direction. $S4$ and $S6$ are convolution in the vertical direction. $S7$ represents the magnitude calculation stage while $S8$ represents the non-maximum suppression. All the stencil parameters are summarized in Table 10.

We evaluate the tuning performance over four fusion combinations of CED with two image sizes (1) 2048 × 2048 and (2) 4096 × 4096. The different fusion combinations are summarized in Table 11, where stencils between () are the fused stencils, stencils separated by the symbol | are the non-fused stencils and number of grids allocated in the shared memory are denoted by $\#shGrids$.

For each fusion combination, we compare the performance of a basic implementation and a tuned configuration for three NVIDIA GPU architectures. The results of the performance comparison on GTX 590, GTX 780 and GTX 750 are represented, respectively, in Fig. 18a–c.

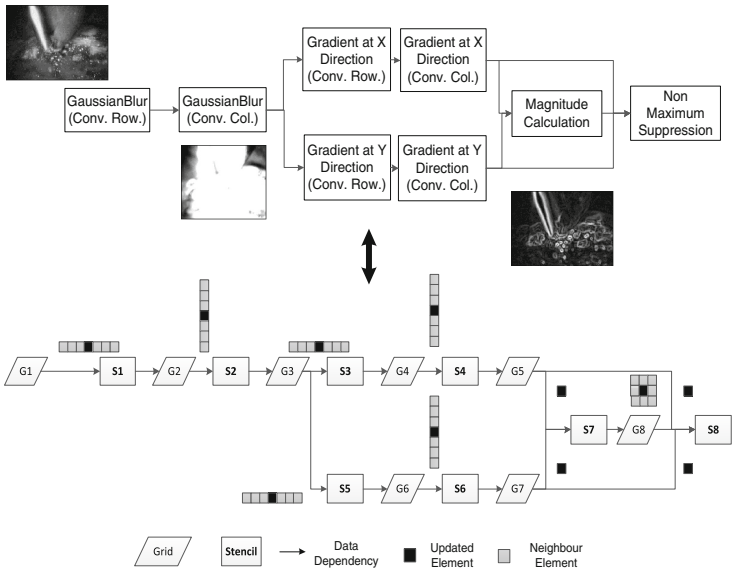


**Fig. 17** Canny edge detection stencils

**Table 10** CED stencils

| Operation | Stencil | $\bigcup_i^{\#inGrid} \{(inGrid_i, H_i)\}, outGrid$ |
|---|---|---|
| Gaussian blur row | $S1$ | $\{(G1, 0 \times 3)\}, G2$ |
| Gaussian blur col. | $S2$ | $\{(G2, 3 \times 0)\}, G3$ |
| Gradient $X$ row | $S3$ | $\{(G3, 0 \times 3)\}, G4$ |
| Gradient $X$ col. | $S4$ | $\{(G4, 3 \times 0)\}, G5$ |
| Gradient $Y$ row | $S5$ | $\{(G3, 0 \times 3)\}, G6$ |
| Gradient $Y$ col. | $S6$ | $\{(G6, 3 \times 0)\}, G7$ |
| Magnitude | $S7$ | $\{(G5, 0 \times 0), (G7, 0 \times 0)\}, G8$ |
| Non-maximum suppression | $S8$ | $\{(G5, 0 \times 0), (G7, 0 \times 0), (G8, 1 \times 1)\}, G9$ |

As first observation, we show that tuned configuration outperforms the basic implementation in the case of several fusion combinations. We can conclude also that the stencil fusion $F2$ provides the highest performance for the three GPUs. This is explained by the fact that $F2$ fuses the largest number of stencils while it allocates the lowest number of grids in the shared memory. The large number of fused stencils improves data locality and avoids the global memory exchanges. Moreover, when the number of allocated grids in the shared memory is small, we can load larger tiles and, as consequence, reduce the number of computation iterations, increasing performance.

Based on these observations and on the tuning performance results, the developer has the ability to investigate efficiently the performance of several number of fusion combinations.

**Table 11** CED fused stencils

| Fused version | Fused stencils | #shGrids |
|---|---|---|
| $F1$ | ($S1$, $S2$)–($S3$, $S4$)–($S5$, $S6$)–$S7$–$S8$ | 2–2–2–0–0 |
| $F2$ | ($S1$, $S2$, $S3$, $S4$, $S5$, $S6$)–$S7$–$S8$ | 2–0–0 |
| $F3$ | ($S1$, $S2$, $S3$, $S4$, $S5$, $S6$)–($S7$, $S8$) | 2–3 |
| $F4$ | ($S1$, $S2$, $S3$, $S4$, $S5$, $S6$, $S7$, $S8$) | 4 |

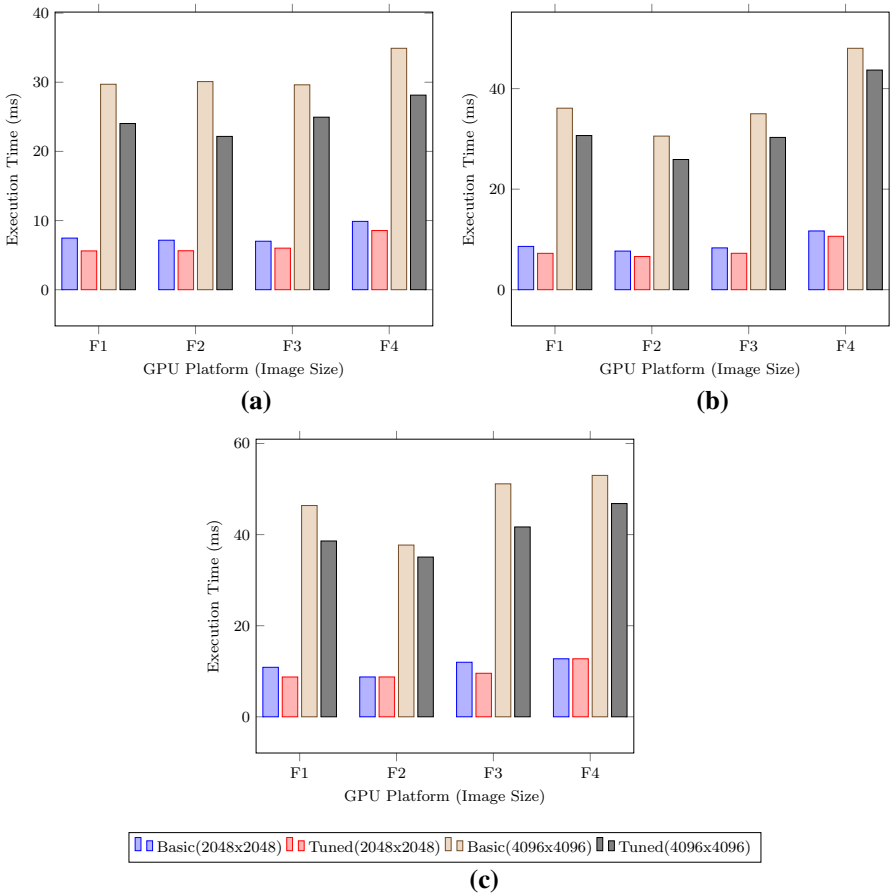


**Fig. 18** Performance comparison of basic and tuned implementation for different fusions on NVIDIA GPUs. **a** GTX 590. **b** GTX 780. **c** GTX 750

## 6 Conclusion

In this paper, we present our methodology for optimizing stencil computations in HPPs. The methodology was the basis of a performance tuning framework that allows multiple stencil computations to be developed.

This framework increases the design space exploration by taking into account characteristics of both the application and the target architecture. The proposed methodology is based on four major steps:

1. program formulation;
2. fusion and tiling configuration;
3. performance model setup, and
4. projection of program characteristics and configurations on performance model.

To automate the process following this methodology, we show a tool that generates all possible implementation configurations on a given GPU specification to provide the resource usage on that architecture. We believe that this feature is very useful for the developer to have an early overview of a given program configuration without the need to implement it on the real hardware. The proposed tool supports also the fusion problem where the developer can test the performance of several fusion combinations and their suitable configurations without writing a complete code for each combination. In order to find a set of optimal configurations, the tool enables the designer to explore the relevant configurations on the proposed performance model. The framework is validated through two concrete image processing applications: (i) the Gaussian blur filter operation, and (ii) the Canny edge detection implemented by performing different fusion combinations. Our implementations target three different NVIDIA GPU architectures. Future work includes improving the data access to the shared memory. Also, we plan to include at least other platforms besides NVIDIA's.

## References


1. Arabnia H (1995) A distributed stereocorrelation algorithm. In: Fourth International conference on computer communications and networks, pp 479–482. doi:10.1109/ICCCN.1995.540163
2. Arabnia H, Bhandarkar S (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–269. doi:10.1007/BF00130109
3. Arabnia H, Oliver M (1987) Arbitrary rotation of raster images with SIMD machine architectures. Comput Graph Forum. doi:10.1111/j.1467-8659.1987.tb00340.x
4. Bhandarkar S, Arabnia H, Smith J (1995) A reconfigurable architecture for image processing and computer vision. PRAI 9:201–229
5. Calandra H, Dolbeau R, Fortin P, Lamotte JL, Said I (2013) Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In: 21st euromicro international conference on parallel, distributed and network-based processing (PDP). IEEE, pp 405–409
6. Cook S (2013) CUDA programming: a developer's guide to parallel computing with GPUs, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA. ISBN 9780124159334, 9780124159884
7. Cuda N (2014) NVIDIA CUDA C programming guide v7.0. Tech. rep. http://www.bibsonomy.org/bibtex/2e90a6474d85eac083c921cf5be29f6ef/toevanen
8. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing. IEEE Press, p 4
9. Djabelkhir A, Seznec A (2003) Characterization of embedded applications for decoupled processor architecture. In: IEEE international workshop on workload characterization (WWC-6). IEEE, pp 119–127
10. Eberhart P, Said I, Fortin P, Calandra H (2014) Hybrid strategy for stencil computations on the apu. In: Proceedings of the 1st international workshop on high-performance stencil computations, Vienna, pp 43–49

11. Grosser T, Cohen A, Kelly PH, Ramanujam J, Sadayappan P, Verdoolaege S (2013) Split tiling for gpus: automatic parallelization using trapezoidal tiles. In: Proceedings of the 6th workshop on general purpose processor using graphics processing units. ACM, pp 24–31
12. Krishnamoorthy S, Baskaran M, Bondhugula U, Ramanujam J, Rountev A, Sadayappan P (2007) Effective automatic parallelization of stencil computations. In: ACM sigplan notices, vol 42. ACM, pp 235–244
13. Luper D, Cameron D, Miller J, Arabnia HR (2007) Spatial and temporal target association through semantic analysis and gps data mining. In: Arabnia HR, Hashemi RR (eds) IKE. CSREA Press, USA, pp 251–257
14. Lutz T, Fensch C, Cole M (2013) Partans: an autotuning framework for stencil computation on multi-gpu systems. ACM Trans Archit Code Optim (TACO) 9(4):59
15. Meng J, Skadron K (2009) Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In: Proceedings of the 23rd international conference on supercomputing. ACM, pp 256–265
16. Pienaar JA, Raghunathan A, Chakradhar S (2011) Mdr: performance model driven runtime for heterogeneous parallel platforms. In: Proceedings of the international conference on supercomputing. ACM, pp 225–234
17. Rahbarinia B, Pedram M, Arabnia H, Alavi Z (2010) A multi-objective scheme to hide sequential patterns. In: The 2nd international conference on computer and automation engineering (ICCAE), vol 1, pp 153–158. doi:10.1109/ICCAE.2010.5451977
18. Tabik S, Murarasu A, Romero LF (2014) Evaluating the fissionfusion transformation of an iterative multiple 3D-stencil on GPUs. HiStencils 2014:81
19. Tang WT, Tan WJ, Krishnamoorthy R, Wong YW, Kuo Sh, Goh RSM, Turner SJ, Wong WF (2013) Optimizing and auto-tuning iterative stencil loops for GPUs with the in-plane method. In: IEEE 27th international symposium on parallel and distributed processing (IPDPS). IEEE, pp 452–462
20. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE (2011) The pochoir stencil compiler. In: Proceedings of the twenty-third annual ACM symposium on parallelism in algorithms and architectures. ACM, pp 117–128
21. Wu H, Diamos G, Wang J, Cadambi S, Yalamanchili S, Chakradhar S (2012) Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: IEEE 26th international on parallel and distributed processing symposium workshops and PhD forum (IPDPSW). IEEE, pp 2433–2442
22. Xu C, Kirk SR, Jenkins S (2009) Tiling for performance tuning on different models of gpus. In: Proceedings of the 2009 second international symposium on information science and engineering (ISISE'09). IEEE Computer Society, Washington, DC, pp 500–504. doi:10.1109/ISISE.2009.60