Ecole polytechnique
Promotion X2012
NICOLET Victor

RAPPORT DE STAGE DE RECHERCHE

A task-based approach to stencil computations

NON CONFIDENTIEL

Département d'informatique	
Champ:	INF591
Directeur de stage :	BOURNEZ Olivier
Maitre de stage :	COHEN Albert
Dates du stage:	23 mars 2015-17 juillet 2015
Nom et adresse de l'organisme :	Département d'informatique Ecole Normale Supérieure 45 rue d'Ulm 75005 Paris France

Abstract

Acknowledgements

I would like to express my gratitude to my internship advisor, professor Albert Cohen, for giving me the opportunity to work in the team, and his advice and discussions. I also want to thank Nhat Minh-Lê for his help and availability with Libkpn and general advice, and also all those who worked on the project, Adrien Guatto and Robin Morisset.
Finally, I would like to thank the whole PARKAS team for their welcome during my stay.

1 Introduction

1.1 Motivation

Multi-core processors with shared memory are now common in devices ranging from mobile phones to supercomputers. There has been a growing interest in providing frameworks to help the common programmer to build efficient programs that would execute on multi-core architectures. Though there have been different approaches, they all try to address the problem of :

- providing the tools to program parts that can be solved concurrently.
- implementing an overall control and coordination mechanism.

Here, we will address computational problems in which breaking the program into parts for concurrent execution is nontrivial. During the computation, time is often spent in nested loops, with iterations depending on each other. When splitting or tiling the iteration space to distribute the work among the different cores, some problems arise :

- the program must stay correct. Breaking a program into parts and modifying its control flow and data flow must be done cautiously. Not all transformations are legal : while some parts of the program can be done in parallel, others must be executed sequentially.
- the architecture has its limitations. We must keep in mind that inter-processor communication is very costly and must be avoided, and that memory usage is a determinant factor in a program's performance.

Previous research has provided many solutions to this problem, aiming to improve parallelism and reduce synchronization and communication overheads. Compilers and domain specific languages can provide automatic parallelization today,thanks to abstractions like the polyhedral model [5], or simpler models with assumptions over the inputs [**?**].
However these solutions are not always optimal. We will provide a simple tiling for time iterated stencil applications that yields good performance on unidimensional jacobi (1-d jacobi), a simple time-iterated stencil.
The scheduling of the different parts of the program is also an interesting field of research. We will compare the performance static scheduling of for loops used in OpenMP to the dynamic scheduling of a task network in Libkpn, the library developed in PARKAS team. The results show that it is possible to equal the performance of hand-tuned OpenMP implementations with the task-based approach using Libkpn.

1.2 Contribution and outline

The PARKAS team, in which I did my internship, is focused on synchronous Kahn parallel networks. There is to main research topics : the definition of high-level languages for embedded systems that provide synchronous data-flow concurrency, and efficient compilation for synchronous and concurrent designs targeting modern shared-memory parallel processors.
I worked more specifically on Libkpn : a library that provides the user lightweight

tasks and first-class dependencies to implement any networks of tasks for parallel execution, based on the model of Kahn process networks [8].
The main purpose of my internship was to show that tasks and dynamic scheduling can perform as well as the static scheduling used in widely used shared-memory models such as OpenMP. Libkpn has already proved its efficiency on classic linear algebra kernels such as Cholesky decomposition and streaming applications. But other real world applications in image processing and scientific computation use computationally intensive kernels.
I initially focused on learning about the state of the art in parallel programming and concurrency models on shared-memory architectures. This included research on program transformations and dynamic scheduling.
In the first application I had to develop, my goal was to use inner parallelism in an image processing application, Harris Corner Detection, but this did not lead to convincing results. Then my focus moved on to a stencil application, 1-d jacobi. We hoped that using tasks and dynamic scheduling, we would improve performance by taking advantage of reuse between successive tiles. My work was an exploration of the potential applications for Libkpn, thus a part of the project and a continuation for previous articles.
The following report presents first the general background necessary to understand the tools and and models I used during my internship. We discuss the results of the experiments with the different Harris Corner Detection implementations. Finally, the main results with 1-d jacobi are presented, with the tiling strategy and task model used.

2 Background

2.1 Stencil applications

Stencil applications are a broad family of compute intensive applications, widely used in the graphic or scientific domain. We can characterize them by an update of a point in a grid, using neighboring points. They have properties that make them a good subject for optimization. We will take as example the jacobi in one spatial dimension as example (see figure 1a).

Characterization A stencil computation is a time-iterated nearest-neighbor computation that operates on each point in a grid and updates each grid point using data in its neighboring points, in recent time steps. We use the stencil characterization presented in [2] :

- Grid points are updated using the values of neighboring points in previous and recent time-steps. Nested loops have always a time-step iteration as outermost loop, and we assume there is no horizontal dependence between statements (the values of a points does not depend directly on the value of other points in the same time-step).
- If we have d spatial dimensions, the computations uses a $d+1$ dimensional grid, the outermost dimension being the time dimension. In the implementation, the grid will be reduced to the spatial grid only, for memory efficiency.

```
for (t = 0; t < T; t++)
   for (i = 1; i < N-1; i++)
       S1: A[t+1][i] = 0.333 *
               (A[t][i-1] +
                  A[t][i] +
                 A[t][i+1]);
```

(a) Perfectly nested 1-d jacobi

```
for (t = 0; t < T; t++)
    for (i = 1; i < N-1; i++)
        S1 : B[i] = 0.3333 *
                (A[i-1] +
                 A[i] +
                 A[i+1]);
    for (i = 0; i < N; i++)
        S2 : A[i] = B[i];
```

(b) Imperfectly nested 1-d jacobi

Figure 1: 1-d Jacobi

- The stencil can always be written in a single-statement of the following form, with a stencil function f :

$$value_p[t+1] = f(value_p[t], value_{neighbors\_of\_p}[t])$$

A stencil computation computes the stencil for each point of the grid over many time steps. Theses applications are quite simple to implement using nested loops, but they are subject to many problems of optimization. These are mainly loop transformations, since the computation is contained in the nested loops.
We will consider the following stencil function f in a one-dimensional grid $\mathbb{G}$, with a data grid $A_{\mathbb{G}}$:

$$\mathbb{G} = [0, N-1] \subset N \ , \ A_{\mathbb{G}} \subset \mathbb{R} \ , \ f(A) = (A[t,i] + [A[t,i-1] + A[t,i+1])/3$$

The stencil computation can be written simply with two nested loops and one statement, using a time-space grid (figure 1a) or two data grids (figure 1b). The use of a bi-dimensional array in the first example with a time dimension allows to nest the two loops perfectly and permit simple transformations. A **perfect loop nest** is a set of nested loops in which all statements appear in the innermost loop. But this is very memory-inefficient, and the time dimension is superfluous.
The second form of the program adds a second statement to swap data in arrays. However, there is more statements and the second loop depends on the first one, creating additional dependencies.

In the remaining of the report, we will consider mainly stencil computations, or at least computations that are structured alike : an outer time loop and dependencies from one time step to another, spanning along the iteration space. The graph will present a time or iterations dimension, representing the domain of the outer loop, and a space dimension for the inner loop.

2.2 Program transformations and polyhedral framework

Most highly computational algorithms are built with a set of nested loop and some statements, accessing and reading data. The program transformations we need to perform concern mostly these loops. In figure 1, we have one outer-loop iterating on time, and an inner loop on the spatial dimension.

```
for (t = 0; t < T; t++)
  for(i = t + 1; i < t+N-1; i++)
     S1 : B[i - t] = 0.3333 *
                  (A[i-1 - t] +
                   A[i - t] +
                   A[i+1 - t]);
```

(a) Skewed 1-d jacobi

```
for(Tt = 0; tT < T; Tt+=64)
  for(Ti = 0; Ti < N; Ti+=64)
    for (t = Tt; t < Tt + 64; t++)
      for(i = Ti; i < Ti + 64; i++)
         S1 : B[i] = 0.3333 * (A[i-1] +
                               A[i] + A[i+1]);
```

(b) Tiled 1-d jacobi

Figure 2: 1-d jacobi after loop transformations

Data dependency In the perfectly nested loop in figure 1a the dynamic instances of statement $S1$ at (i, t) reads data that has been written by $S1$ at $(t1, i)$, $(t1, i1)$ and $(t1, i + 1)$. We call this a *data dependency* : a dynamic instance of a program statement refers to data of a preceding statement. We will refer simply to a *dependency* since we are not interested in other kinds of dependencies.

2.2.1 Loop transformations

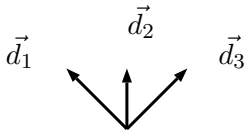
To exploit parallelism in the application, we have to modify the nested loops structure, while ensuring the program still produces the correct output. We present some correct loop transformations [1] to modify the program and produce the correct strategy.

- loop skewing can be useful to enable parallelism. Since it only changes loop bounds and alters accordingly the use of index variables, it is always legal to perform skewing on loops. Figure 2a shows a skewed version of 1-d jacobi.
- loop tiling is useful to improve cache performance. When performing loop tiling on an outer loop (containing other loops), we divide the iteration space into chunks and iterate over these parts. Figure 2b show a tiling using tiles of size 64×64.
- loop distribution (or loop splitting, loop fission) can be used to distribute a single loop into different loops. This can create perfect nested loops, or reduce the dependences in the nested loops. The inverse transformation is loop fusion.

Loop tiling and distribution legality is constrained by the dependences. We will speak of tile dependencies when we perform tiling, and see that loop transformations alter theses dependencies. Yet these transformations can be expressed in a more high level model, which will also allows to find suitable transformations for parallelization. Programming manually transformed loops is a very tedious work and error prone. During my internship, I programmed various tilings using loop transformations, but the goal today in high performance computing is to build an automatic system using the model described in the following paragraph. Though I did not use it myself except for some experiments with Pluto [5], I could not dissociate the optimization of iteration

$$D_{S1} : \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ t \end{pmatrix} \geq \begin{pmatrix} 1 \\ -(N-1) \\ 0 \\ -T \end{pmatrix}$$

(a) 1-d jacobi original polytope



(b) 1-d jacobi dependence vectors

spaces and tilings from this field of research, also very important in the team where I was working.

2.2.2 Polyhedral model

Simple transformations are easy to achieve without excessively complexifying the original program. But when loop nests become more complicated, a mathematical framework can prove useful to express loop skewing, tiling, distributing etc. We will express the properties of our stencil applications in a linear-algebraic representation of programs, the polyhedral model, and provide a brief description of it. Given a representation of a program, it will be easier to express further transformations and show their validity. This model has been proved very useful in automatic parallelization at compile time [3], because it provides many abstractions to reason about program transformations.

Polyhedral representation of programs A program is a set of dynamic instances of statements. Each instance S is defined by its iteration vector $\vec{i}$ containing the values of the indexes of the loops containing the statement. When inner loop bounds depend on outer indexes values, the set of iterations vectors representing different instances of a statement define a polytope D_s (see figure 3a), the domain of the statement, with m_s its dimensionality. In this model, we consider only convex iteration spaces.

Polyhedral dependencies The difficulty of parallelizing stencil applications arises when considering the dependencies between dynamic instances of the statements. Depen- dences are affine and exact, and only carried by the time loops in our stencil applications. The data dependence graph is a directed multi-graph, with each node S_i representing a statement and each edge $e \in E$ representing a dependence between statements S_i and S_j . e can be represented here a constant distance vector by analyzing the source and target iterators in the domain D_s . In 1-d jacobi we have only 3 dependencies $\vec{d_1} = (1, 1)$, $\vec{d_2} = (0, 1)$ and $\vec{d_3} = (1, 1)$ (figure 3b

2.2.3 Tiling

Tiling in stencil applications is a key transformation to improve locality,memory reuse and parallelism. There has been a considerable amount of research in this domain, and we present here a summary of previous work.

Tiling legality In figure 1b we have an example of imperfectly-nested loops. It raises more problems when searching for valid tiling hyperplanes. Using the polyhedral framework, we have conditions provided in prior research, expressed in [5]. A legal tiling is a tiling that ensures that we can construct a total order between tiles for execution (there is no tiles that depend on each other) and provided inter-tile dependencies are satisfied, a tile can execute atomically.

Definition 1. *A hyperplane for a statement S_i is of the following form:*

$$\phi_{S_i}(\vec{x}) = \vec{h} \cdot \vec{x} + h0$$

where $\vec{x} \in D_s$ is an instance of the statement S_i, $h0$ is the translation and $\vec{h}$ is the hyperplane normal vector.

Tiling legality 1. *Let ϕ_{S_i} be a one-dimensional affine transform for statement S_i. For $\phi_{S_0}, .., \phi_{S_k}$ to be a legal tiling hyperplane, the following should hold for each edge e from S_i to S_j :*

$$\phi_{S_j}(t)\phi_{S_i}(s) \geq 0 \ , \ P_e$$

In our example with only one statement, a legal tiling hyperplane with normal vector $\vec{h} = (h_i, h_t)$ must satisfy the following conditions :

$$\vec{h} \cdot \vec{d_1} = h_t h_i \geq 0 \ , \ \vec{h} \cdot \vec{d_2} = h_t \geq 0 \ , \ \vec{h} \cdot \vec{d_3} = h_i + h_t \geq 0$$

Here rectangular tilings are not considered legal, unless a space transformation is performed before. Parallel computations should minimize interprocessor communications and rectangular tilings are not satisfactory when dependence vectors have negative components along the space dimension.

Tiling for locality The spatial dimensions of most problems involving stencil computations are usually quite large, often larger than the cache sizes or other on-chip memory size. Parallelism is present along the spatial dimension, but tiling is necessary to improve data locality during the computation and the different memory speeds.
The objective here is to fit the tile spatial domain into on-chip memory, potentially at different levels. The tiling concerns here the inner loops.

Tiling for reuse The goal is to work around the problem of limited memory bandwidth by using present memory as mush as possible before releasing it to memory levels with longer access delays. Many computations are throttled by this issue, an efficient schedule along the time dimension can dramatically improve the performances. Tiling only the inner loops to provide locality is not sufficient. But transforming the nested loops including the outer one can enhance memory reuse between different tiles.

Tiling for communication-free parallelism In current multiprocessor architectures, communication between processing units is costly and has to be avoided. Tiling for parallelism implies finding calculations that can be executed at the same time and without data from other concurrent calculations.

A tiling enables a tile-wise concurrent start if all that tiles along one dimension of the iteration space can be started concurrently. If tiles carry dependences along one dimension, they cant be started concurrently along this dimension. When dependences span the entire iteration space, there is often a trade-off between locality, memory reuse and and communication-free parallelism.

2.2.4 Tiling techniques in previous research work

Skewed tiling Skewing the iteration space is a standard tiling technique for time- iterated stencils along with rectangular space tiling. Visually speaking, figure 4 presents the shape of the tiles, parallelograms along the time dimension, and regular and legal tiling long the spatial dimensions. This improves memory reuse but creates inter-tiles dependencies along the space dimension, restricting parallelism to tile wavefronts along diagonals, and forbidding a concurrent start.

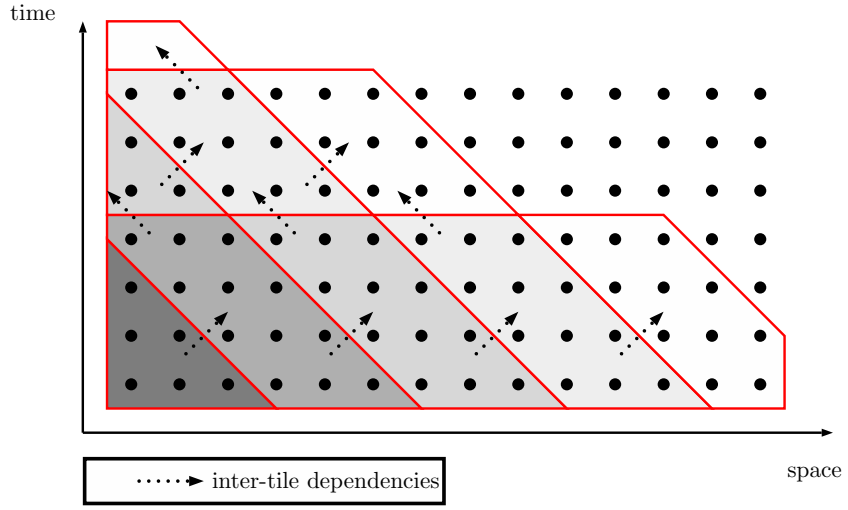


Figure 4: Skewed tiling with inter-tile dependences and wavefront

Overlapped tiling Using overlapped tiling enables a concurrent start among the spatial dimension and removes synchronization between tiles. But it also causes a significant amount of redundant computation, especially if the time dimension is important. Moreover, redundant computations often need to be stored in shared memory, causing another overhead. Figure 5 shows schematically the shape of the tile along on spatial dimension and the time dimension. The shape of the tile is defined by the dependencies projected on the spatial dimension. Thus, overlapped tiling can be very efficient on low-order stencils and few iterations.
Hierarchical overlapped tiling [11] has proven efficient for stencil computations by balancing communication overhead and redundant computation cost.

Split tiling Split tiling [6] does not make redundant computation while allowing a concurrent start along the space dimension. In figure 6, the lower darker tiles are executed concurrently in a first step, then the lighter tiles, mapped onto threads so to reuse a maximum amount of memory. The horizontal double line represents the step by step calculation : between each level

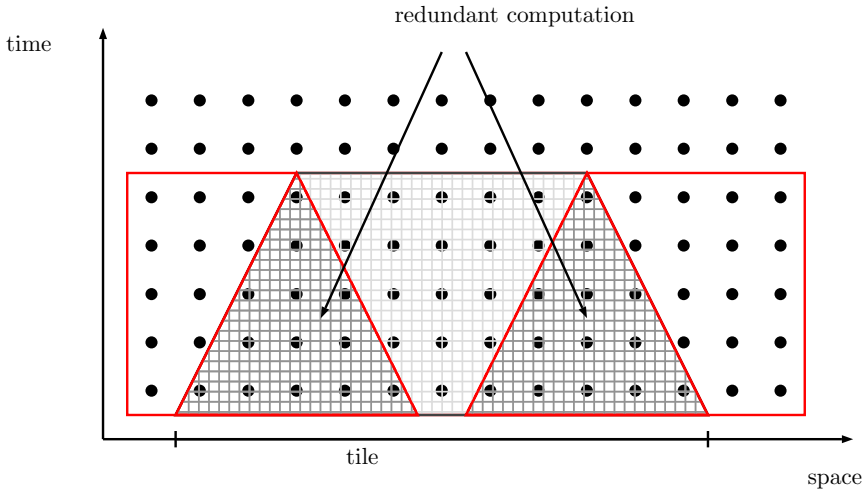


Figure 5: Overlapped tiling with inter-tile dependences and redundant computations

and between light and dark tiles, we have a barrier. However, we could schedule the computation differently, using tasks where we wait for dependences to be satisfied. The new tiling, diamond tiling, shows a more natural space partition to use a schedule without space-wide barriers.

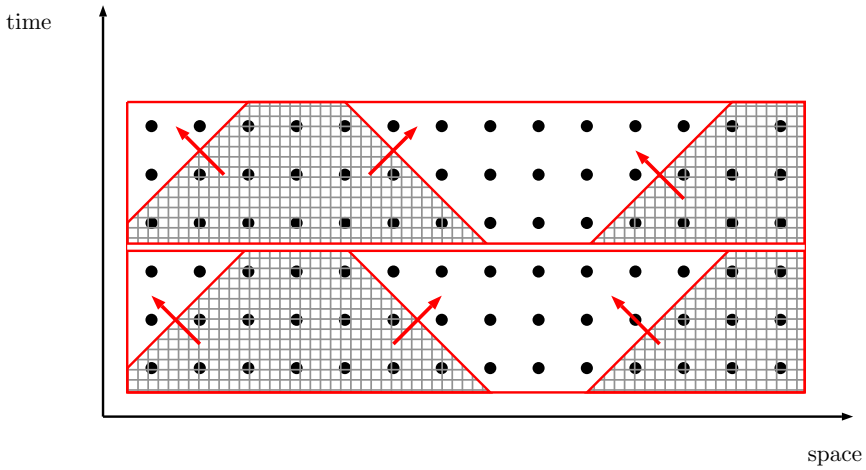


Figure 6: Split tiling with inter-tile dependences

Diamond tiling Diamond tiling [2] allows a concurrent start along space dimensions, and a good schedule can improve memory reuse between successive levels of tiles. But with this tile shape, the tiles of same shape are different regarding the integer point the contains, thus the computation they contain. This has a small impact with control divergence : the computation varies depending on the tile.
Hybrid hexagonal tiling [**?**] tries to address this problem by computing a regular tile shape based on diamond tiling.

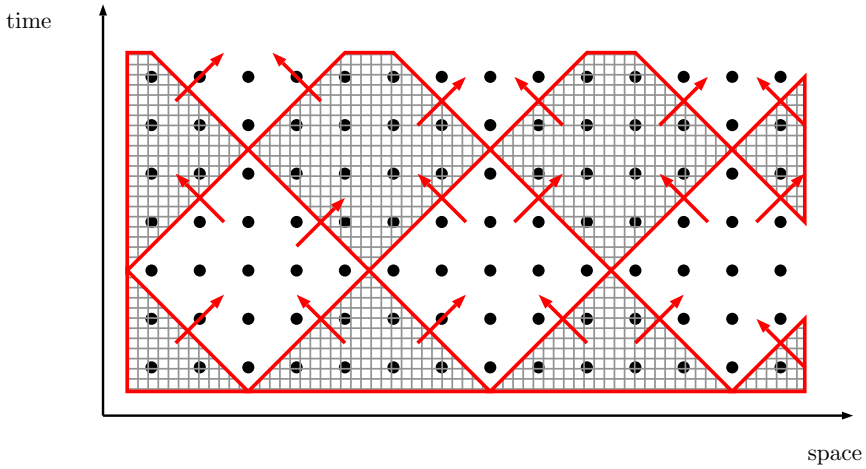


Figure 7: Diamond tiling with inter-tile dependences

2.3 Scheduling

Scheduling a parallel computation refers to the problem of assigning work to each processor. A good schedule must ensure load balancing, data locality and reuse along time. Tiling already provides the locality, assuming one tile is processed by only one core and the tile has the fitting size. But inter-tile memory reuse and load balance is achieved by scheduling correctly how tiles are computed (in which order, on which core). In the exploitation of loop level parallelism, manually scheduling the iterations can be a difficult task for the programmer. In the shared-memory programming model OpenMP, there is typically two scheduling strategies : static or dynamic.

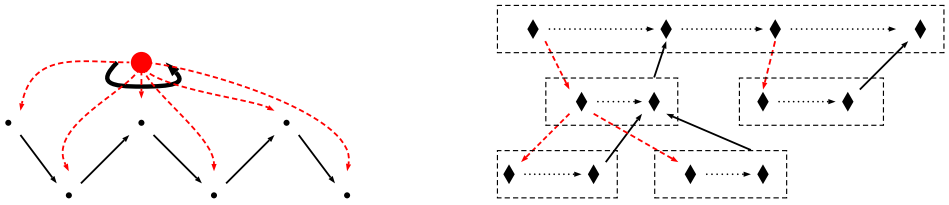
2.3.1 Static loop scheduling

For loops with good balance among iterations are good candidates for a static schedule, since a simple splitting of the loop work will yield balanced partitions of the total computation. Assuming the thread pool has n_w worker threads, a static schedule evenly divides the loop iteration space into n_w chunks, and assigns each chunk to a worker thread. In most cases in our experiments, n_w is equal to then number of cores available.
To parallelize an already tiled computation, the static schedule is applied to the outer loops iterating over the tiles. Without additional efforts or an advantageous configuration, the static schedule will not ensure data reuse between tiles (assuming there is a schedule providing reuse). If we take a set of $N\times$ tiles, considering the space dimension has been divided into N tiles and we have T steps, with a constant dependency vector $(0, 1)$ (the tile depends on the tile in the previous time step only). If we naively parallelize the outer spatial loop, we assign at each time step a chunk of size $c \leq N$ to each core, without any consideration of total chunk size. If the size of c tiles is greater than the cache size, the memory of the first tiles will be flushed out from the cache. But using another schedule, this memory could be reused by the tile at the next time step. In this trivial example, we show that with an additional effort we can improve reuse. But using dynamic scheduling this task can be

avoided.

2.3.2 Tasks and Libkpn

In [8] G. Kahn defines a language for parallel programming, viewing parallel programs as a networks of computation units, the tasks. These wrap parts of the program meant for concurrent execution.
A parallel program is an oriented graph, were nodes are the tasks (or processes) and edges communication pipes. There is additional edges that represent input and output of the program. Each process computes a sequential program, receiving data only from its incoming edges and issuing data on his outgoing lines. In the following paragraph, we distinguish the data-flow dependencies introduced by Kahn and the creation dependencies. The first imply a partial execution order. A task cannot execute before all its incoming dependencies have not been satisfied. Also a node cannot wait or produce data on more than one channel simultaneously.
Creation dependencies represent the parent-child relations between task. There is also an order between sibling task since the spawning or creation of child tasks is sequentially ordered in the parent task.
We distinguish here the task graph, containing task and pipes, from the creation tree. In previous implementations of task parallelism languages such as Cilk and OpenMP 4, data-flow dependencies are restricted to tasks that are directly related in the creation tree or sibling tasks, depending on the model. On the contrary, Libkpn allows arbitrary dependency creations betweens tasks
.
Two models can be distinguished in previous languages and frameworks. We illustrate these two models in figure 8. Dashed arrows represent the creation tree, whereas full arrows are data-flow dependencies. In OpenMP and SMPSs (figure 8a, the dependencies are built on sequentially ordered memory descriptors. A master task (top dot) creates all its children sequentially. The execution order of its children is determined by the data dependencies.
In the Cilk-like models of multithreaded computations, the creation tree is not limited to one task sequentially spawning the task graph, since each process can spawn children. The limitation here is in the fact that a parent can only wait on its child. In this model of multithreaded computation in figure 8b, threads (diamonds)are non-blocking function and cannot wait but instead spawn successor threads (dashed arrows). Then the successors wait for the data dependency to be satisfied. The procedures enclosing the threads (dashed rectangles) are sequentially executed parts of the program, similar to tasks. In Libkpn, in cyclic task graphs or error-prone implementations, but any process network can be implemented. Data consumer-producer relationships are represented as first-class objects that can be passed between tasks. A channel between two tasks can be created without knowing the producer or the consumer. In figure 9, when the bottom task is executed, the consumer task may not have been created. The dependencies here do not follow the two-branch creation tree.



(a) Master task spawning task graph (b) Cilk model of multithreaded programs

Figure 8: Different models for task parallelism languages

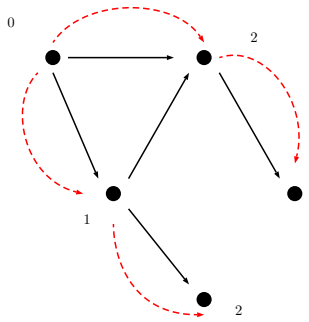


Figure 9: Arbitrary dependencies and creation tree

2.3.3 Dynamic task scheduling

While some systems offer compile-time scheduling [10], a natural approach to task-based parallelism is scheduling at runtime to adapt to dynamic data dependencies and unknown hardware platforms. Tasks can be in three states : either running, ready when all the dependencies have been satisfied, or stalled, when the tasks waits for data. Libkpns relies on a work-stealing scheduler with additional mechanisms. The work-stealing algorithm [4] manages the transfer between the set of ready and running tasks. The additional mechanism is needed to deal with the stalled tasks.
During its execution, a task can :

- spawn another task.
- die, when its execution terminates.
- stall, if it waits for a dependency to be satisfied.
- enable another task, by satisfying the dependency on which it stalled.

Work-stealing scheduler The work-stealing scheduler outlined here schedules the execution of a set of tasks with a dependency graph on a parallel computer. Each processor has a ready deque containing tasks ready to be executed, with a top and a bottom. Tasks can be inserted at the bottom of the deque, and removed at either end. A processor treats its ready deque as a call stack : it removes the task at the end to get work, but other processors can steal tasks from the deque by removing the top task.
A processor P begins to work by pulling the bottom task T_a of its ready deque, and executes it until T_a spawns, stalls, dies or enables another task :

- if T_a spawns a task T_b, then T_a is placed at the bottom of the ready deque of P and P starts executing T_b.
- when T_a stalls, P checks its ready deque and begins work on the bottom task. If the queue is empty, it tries stealing the top task of a randomly chosen processor.
- when T_a dies, it follows the same rule as when T_a stalls.
- if T_a enables another task T_b, T_b is placed at the bottom of the ready deque of P .

Here when a processors attempts to steal work from another processor, it choses his victim randomly. The attempt to steal can either fail, is the deque of the victim is empty, or success. If it fails, the processor chooses another victim at random, and attempts stealing again, until it finds work.
This algorithm ensures some executions properties with bounds on memory and time. If we call t_1 the minimum serial execution time of the multithreaded computation on a single processor and t_∞ the minimum execution time on an infinite number of processors, the execution time of the multithreaded program on p processors is bounded by $t_1/p + \bigcirc(T_\infty$. The space requirement is also bounded by pS_1 with S_1 the space required by the computation on a single processor. This means that there is no space overhead coming from the multiprocessor execution, and the time overhead is bounded.

Libkpn scheduler Libkpn scheduler is a work-stealing algorithm with some adaptations. The inner mechanisms of the scheduler have not been described above, but in multiprocessor environment, task-scheduler synchronization is needed when any user-space execution hands back control to the scheduler. In this case Libkpn relies on relaxed C11 primitives. The internal algorithm, managing the tasks cycles is correct but yields better performance since it gets rid of most of the heavy acquire/release lock synchronizations that would be used in a naive implementation. Chase and Lev (2005) have proposed a lock-free deque mechanism, leading to an implementation on current multicore processors by L al. (2013).
Libkpns scheduling algorithm itself is based on work-stealing with some adaptations. Since it works with tasks and there is a possible communication between them, there is also a need to add a mechanism to manage task stalling, since the previous was designed to handle only running and ready tasks. This algorithm has also been implemented with single-producer single-consumer lock-free queues to manage dependencies.
Since my internship is not focused on the scheduling but in implementing applications that use this scheduler, I will not dwell on this subject detailed in ** PAPER? **. But the understanding of the tool I was going to use seemed quite necessary to catch the differences between Libkpn and other tools.

3 Harris Corner Detection

In the first part of my internship, I had to explore the possibility of using tasks in OpenMP for Harris Corner Detection pipeline [7]. Corner detection

is widely used in computer graphics, and as a comparison point we took a state-of-the-art compiler and domain specific language for image processing pipelines [9].

3.1 Harris Corner Detection pipeline and parallelization opportunities

The algorithm of Harris Corner Detection is an image processing pipeline divided in several elementary steps involving simple computations or stencil computations :

- Compute I_x and I_y derivatives of the image along x and y directions.
- Compute the product of derivatives at every pixel I_{xx} , I_{xy} and I_{yy}.
- Smooth the image, calculating at each point the mean of the neighboring points : S_{xx}, S_{xy} , S_{yy} .
- The output is defined as follows :

$$H = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix} \ , \ Harris = Det(H) - k * Trace(H)^2 \ , \ k \in \mathbb{R}$$

The pipeline graph in figure 10 shows how the different steps are linked together, and the parallelization opportunities in the pipeline. For example, steps I_{xx}, I_{xy} and I_{yy} can be executed in parallel and only depend on I_x and I_y . In computer vision, the size of the images is often far greater than the sizes of fast memory. To make an efficient use of on-chip memory and to expose more parallelism, tiling the spatial dimension is also useful. Because of the stencils in the pipeline, there is need for communication between tile, thus a careful scheduling.
In the implementation from Polymage, the pipeline parallelism is not exploited. The tool uses a heuristic for grouping different stages of the pipeline and use overlapping tiles in the computation.
In the task approach, the goal is to get rid of redundant computation and to expose more parallelism in the pipeline.

3.2 Algorithm with tasks

In order provide more parallelism in the pipeline, we implemented Harris Corner Detection using both the tiling extracted from the output of Polymage and task for the stages of the pipeline. In Polymage's output, the image is partitioned into rectangular tiles of size 32×256. The pipeline can be executed atomically on each tiles :

- the tasks below I_x and I_y are fused into one task (the inner loops in the tile are fused).
- the pipeline is executed sequentially in each tile.
- tiles are overlapping to avoid the need to synchronize during the pipeline.

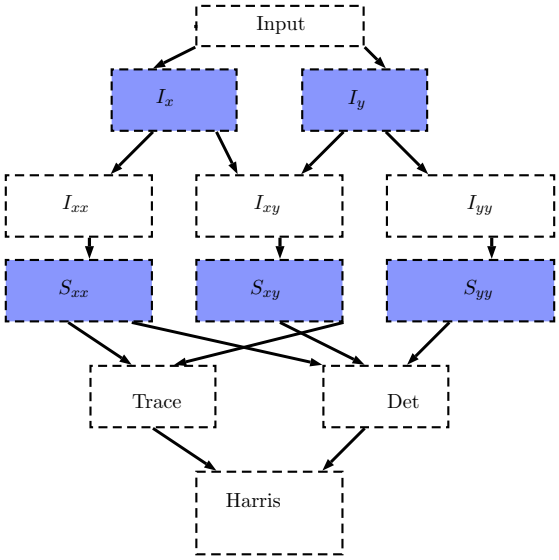


Figure 10: Harris Corner Detection image processing pipeline

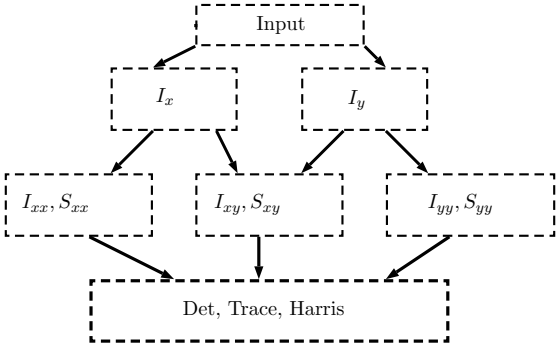


Figure 11: Harris Corner Detection with fused stages

In our approach using tasks, we want to implement the algorithm and keep parallelism in the pipeline. But some steps are executed sequentially (I_{xx} and S_{xx} , I_{yy} and Syy , etc.), therefore we grouped them . We also grouped the three last stages, given their weak computational weight to balance the different tasks. In each task (I_{xx}/S_{xx}) we fuse the loops to produce only two nested loops with one statement. The task graph in figure 11 shows the modified pipeline.
For each spatial tile in the image, we create the set of tasks corresponding to the pipeline with their dependencies, both intra-tile and inter-tile. We do not require tiles to execute atomically anymore : tiles do not overlap but the is synchronization between neighboring tiles. It is the schedulers job to use memory efficiently in consideration of the dependences.

3.3 Results

The benchmarks were run on an Intel Xeon E5-2630 with 12 cores (24 hyperthreads) and 15Mb cache, and a dual-core Intel Core i7-4510U with 4Mb

cache, and we used `gcc 4.9.1` and `icc 15.1` to compare versions from different compilers with 3 levels of optimization.

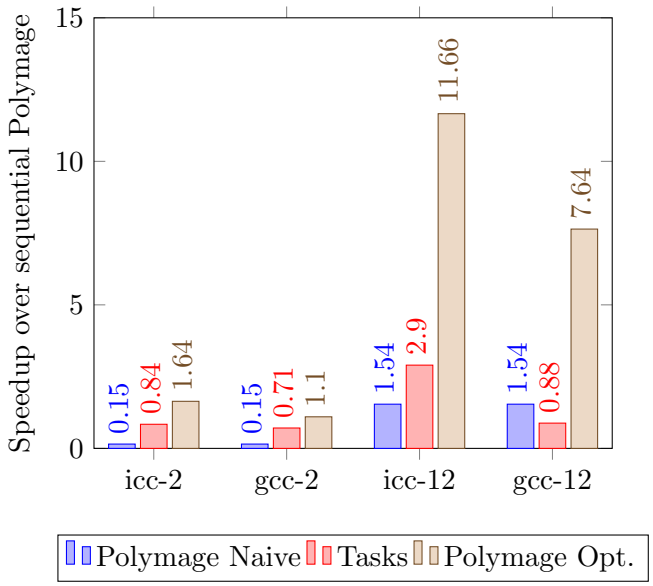


Figure 12: Speedup relative to sequential Polymage, on machines with 2 and 12 cores, with gcc4.9.1 and icc 15.1

The algorithm using tasks does not perform as well as the optimized version from Polymage. The gain from the absence of redundant computation, and a potential good reuse of memory in task scheduling does not compensate the loss due to more synchronization and fused-loops distribution. But the excellent performance of Polymage, nearly scaling linearly with the number of cores (see [11] for more results) also relies on good compiler optimizations of the heavily computational step resulting from loop-fusion.
We tried to measure performance without advanced compiler optimizations, using `-O2` for `gcc` and `icc`. This disables automatic loop vectorization. Since most of the time is spent in the computational fused loops, performance is badly reduced and parallel implementations of Harris Corner Detection are slower than the sequential with automatic vectorization. But the version using OpenMP tasks performs better without compiler vectorization than Polymage version using heavily fused loops.
The different implementations are available on Github at :
`https://github.com/victornicolet/harris-corner-implementations`,
with both the sources from Polymage and the algorithm using tasks.

4 1-d Jacobi

Time iterated stencils can be good candidates for an efficient use of tasks. Here we are interested in small iteration numbers, and we implement a specific tiling strategies in this case.
The OpenMP implementations are available at
`http://www.github.com/victornicolet/tiling-strategies`,
along with some results.

4.1 Half-diamonds tiling

Diamond tiling has proven an efficient tiling strategy since it provides a concurrent start, locality and possible reuse. In out case, we consider only cases when the number of iterations is low. The iteration space is tiled using hyperplanes $(1,1)$ and $(1,1)$ as shown in figure 13. We easily show the legality of this tiling since with the provided hyperplanes. For each statement S with the tiling hyperplanes for each dependency $d \in (1,1),(0,1),(1,1)$ we have $h1 \cdot d \geq 0$ and $h2 \cdot d \geq 0$.

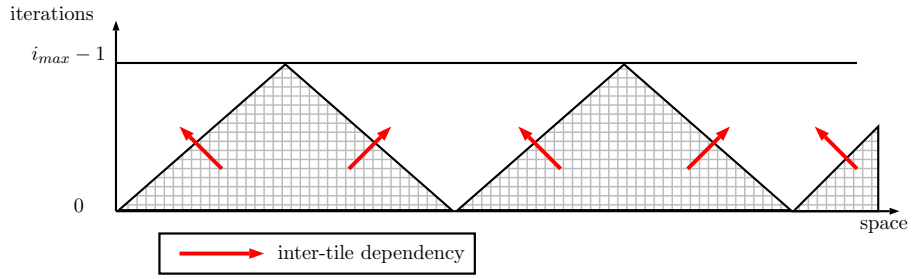


Figure 13: Half-diamonds tiling for low-iterations stencil computations

We have implemented a first naive version, using a global barrier between lower and upper tiles, and static scheduling. Even if this tiling seems quite simple, hand-writing tiled loops is quite tricky. This version has a few errors on the borders of the array, but these errors should not affect performance in any way.

Grouping tiles take advantage of the memory architecture and fast L1 caches, it can be more interesting to adjust the size of the chunks in the loops, and the execution schedule. In the previous version, if the data size is significantly greater than the lower caches, the memory of the first lower tiles will be flushed out the caches before being reused by the upper tiles. To solve this problem, we group loop parts together, adjusting their size proportionally to the number of physical cores of the platform. The goal is to take advantage of static scheduling, which distributes evenly chunks of the loop across processing units to specify exactly how much computation we want to distribute. The group size is calculated depending on the number of processors n_{cores} , the size of the L1 cache and the size of the tile in memory (the size of the array containing the temporary elements between lower and upper tiles) :

$$Group\ size = n_{cores}(L1\ cache\ size)/(tile\ size\ in\ memory)$$

Results The tiling for low iteration numbers without grouping tiles has proven efficient, scaling well with the number of processors. But the expected gain with grouping tiles has not been observed. The version with grouped tiles performs similarly to the simpler version, with a constant overhead (see figure 14). The schedule on the implementation with a global barrier between lower and upper tiles is already taking advantage of reuse through a good mapping

Figure 14: Speedup relative to sequential version, on Intel Xeon E5 with 12 cores

of tile on the threads.

We still need to determine why grouping the tile into a size fitting the fastest cache level does not improve the performance, since it should improve reuse. Varying the group size does not affect performance either, even if the tile scheduling on the different processor is as expected : in each group upper tiles are executed on the same core as the lower tiles at their right.

4.2 A task-based approach

Using a task based approach, we wanted to show that the scheduler can improve data reuse in caches as much as the static scheduling with grouping. We constitute tasks containing one lower tile and the upper tile immediately to its left. The choice of taking the left tile will reduce task stalling. Indeed, the data dependency will probably be satisfied by the left task when the current task will begin to compute its upper tile. In figure 15 we summarize the task graph construction: each tasks spawns its successor, then computes the lower tile and waits at the barrier that the left task has finished to compute the upper tile. The upper half-tiles at the borders of the domain are treated separately.
Data reuse can be exploited at the border between upper and lower tiles, by executing neighboring tasks on the same processor. But parallelism should not suffer and the computation load has to be balanced between processing units.

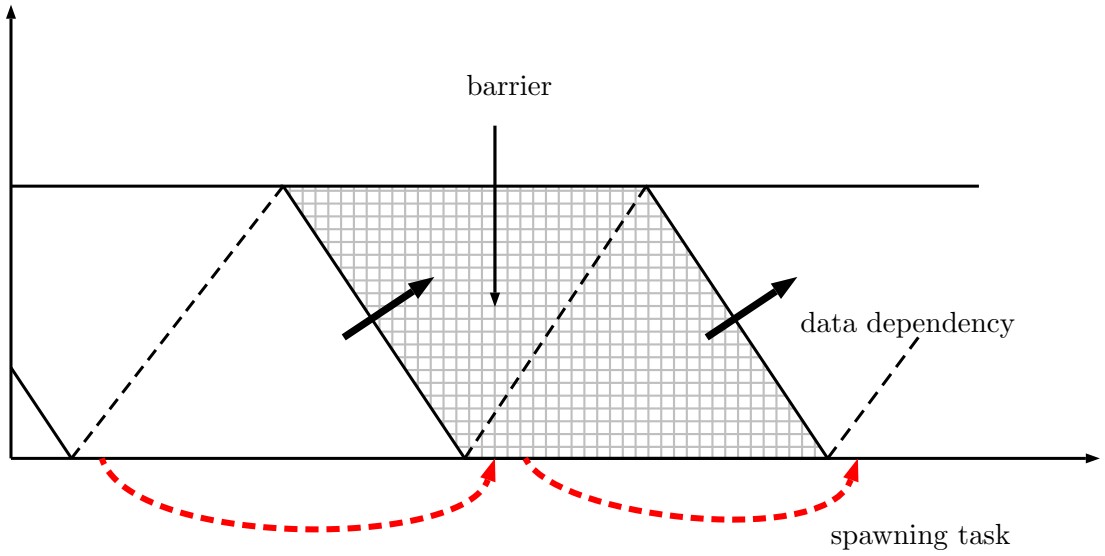


Figure 15: Tasks dependencies, synchronization points and creation schema

4.3 First results

This first version uses fixed-shape tiles : half diamonds with fixed hyperplanes, only the width varies depending on the number of iterations. It does not perform well compared to the statically scheduled OpenMP reference we use with the same tiling. On small iteration numbers the task version is slower than the sequential version. It can be explained by the granularity of the execution, the tasks are very small given the small iterations numbers. The time spent in the computation is mostly scheduling: the runtime manages many tasks, which terminate very rapidly.
The performance of the scheduler measured with this implementation with such very small tasks was still better than OpenMP4s using tasks pragmas.

4.4 Tile shape determination at runtime

The previous results showed that the runtime was under performing because of the high granularity of the task graph. Since our tiling was designed for small iteration numbers, the tile sizes were relatively small in terms of memory occupation and computational weight.
With tiles of variable size shaped at runtime, depending on the number of iterations, we can increase the weight of this tasks. Similarly to the computation of group size in the previous section,the tile size depends on the size of the cache to ensure a good locality, and that sufficient time is spent in the task. We vary the orientation of the oblique hyperplanes with a slope between 1 (as in the previous version) and a value so that there is enough tiles to be distributed among the working threads. Tiling hyperplanes normal vectors are now $\vec{h1} = (a, 1)$ and $\vec{h2} = (a, 1)$ with $1/a = n_{cores} \times (L1\ cache\ size)/(tile\ size\ in\ memory)$. This tiling is still legal and valid with $a \leq 1$, the scalar product between dependencies and hyperplanes normal vectors is positive or zero.
n The only change using this tiling is the amount of computation in each tile. We can fear that using this skewed tiling space with large strides in the loop could lead to worse compiler automatic vectorization.
This technique combined with the task approach to this problem has given better results, in par with OpenMP on small problems but performing better with increasing sizes, showing speedups better than linearly-increasing with the number of cores. On the 12-core machine with a problem size of 32 MB, we reached speedups up to 20 times better than sequential, 3 times better than OpenMP. The results are shown in figure 16.
We have bad performance with small iteration sizes and small space sizes. This may be coming from the lack of parallelism with a too-small number of tasks because the slope of the sides of the tiles was not bounded : for small iterations, the tile is very wide and it could span a large part of the space. The final program at the moment I am writing this report has still a few errors concerning the correctness of the output, but they do not affect performance in any way. They come from hand-tuning the loops and the tiling, a process that we seek to automate through polyhedral code generation tools and compilation. The main goals here were to show a well performing tiling for time-iterated stencils, and validate the approach of implementing the algorithm using lightweight task and dynamic scheduling instead of the classical approach with static scheduling through loop-chunk distribution.

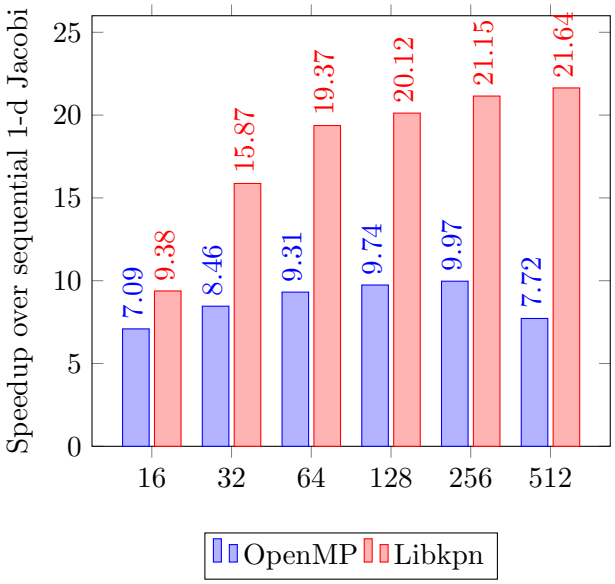


Figure 16: Speedup relative to sequential 1-d jacobi on 12 cores, with different iteration sizes on a 32 Mb problem.

5 Conclusion and future work

During my internship, I have had the opportunity to apply and augment my knowledge in computer science. I have learned a lot about the subjects mentioned in this report, from the tilings and the polyhedral compilation problems, to the Kahns parallel tasks network. During the first month of my internship, I mostly studied the topics I did not master through a bibliographic research on both recent and older publications. The first application did not provide satisfactory results but prepared me for the second subjects, especially on mastering profiling and debugging tools adapted with parallel applications. Then after a seminar from Uday Bondhugula, we discussed the possibility of taking advantage of more parallelism in one of the applications presented, Harris Corner Detection, without convincing results. Then my focus moved on to a more simple example, 1-d jacobi, using a partly new tiling strategy for small iterations and implementing an application for Likbkpn, the library developed at Parkas. At the time of writing this report, my intern- ship is not yet finished. There is plenty applications to implement to show the strength of the library but since I have few time left, I will concentrate on improving the current example.
Optimization is a very experimental field, and I learned to use various tools to validate or refute the hypothesis made during the process.

6 References

References

[1] Bacon, D. F., Graham, S. L., and Sharp, O. J. Compiler Transformations for High-performance Computing. *ACM Comput. Surv. 26*, 4 (Dec. 1994), 345–420.

[2] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 40:1–40:11.

[3] Bastoul, C. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2004), PACT '04, IEEE Computer Society, pp. 7–16.

[4] Blumofe, R. D., and Leiserson, C. E. Scheduling Multithreaded Computations by Work Stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[5] Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, L. Hendren, Ed., no. 4959 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 132–146.

[6] Grosser, T., Cohen, A., Kelly, P. H. J., Ramanujam, J., Sadayappan, P., and Verdoolaege, S. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (New York, NY, USA, 2013), GPGPU-6, ACM, pp. 24–31.

[7] Harris, C., and Stephens, M. A combined corner and edge detector. In *Alvey vision conference* (1988), vol. 15, Citeseer, p. 50.

[8] Kahn, G. The Semantics of a Simple Language for Parallel Programming, 1974.

[9] Mullapudi, R. T., Vasista, V., and Bondhugula, U. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 429–443.

[10] Thies, W., Karczmarek, M., and Amarasinghe, S. Streamit: A language for streaming applications. In *Compiler Construction* (2002), Springer, pp. 179–196.

[11] Zhou, X., Giacalone, J.-P., Garzaran, M. J., Kuhn, R. H., Ni, Y., and Padua, D. Hierarchical Overlapped Tiling. In *CGO '12 Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 207–218.