The Journal of Supercomputing (2019) 75:1382–1395
https://doi.org/10.1007/s11227-018-2638-5

High-performance code optimizations for mobile devices

Sergio Afonso[1] · **Alejandro Acosta**[1] · **Francisco Almeida**[1]

Published online: 11 October 2018

Abstract
Mobile devices have seen their performance increased in latest years due to improvements on System on Chip technologies. These shared memory systems now integrate multicore CPUs and accelerators, and obtaining the optimal performance from such heterogeneous architectures requires making use of accelerators in an efficient way. Graphics Processing Units (GPUs) are accelerators that often outperform multicore CPUs in data-parallel workloads by orders of magnitude, so their use for image processing applications on mobile devices is very important. In this work we explore tiling code optimizations for GPU applications running on mobile devices. A dynamic adaptive tile size selection methodology is created, which allows finding at runtime close-to-optimal parameterizations independently of the underlying architecture. Results demonstrate the performance benefits of these optimizations over a set of stencil-based image processing benchmarks.

Keywords Auto-tuning · GPGPU · OpenCL · Android · Heterogeneous architecture

1 Introduction

Handheld devices performance has quickly improved over the last decade, mostly thanks to the development of System on Chip (SoC) technologies, powered by the

This work was supported by the Ministry of Science, Innovation and Universities through the project TIN2016-78919-R and the Grant Number FPU16/00942, by the Government of the Canary Islands through the project ProID2017010130, by the CAPAP-H network and by the cHiPSet COST Action.

✉ Sergio Afonso
safonsof@ull.es

Alejandro Acosta
aacostad@ull.es

Francisco Almeida
falmeida@ull.es

[1] Department of Computer Engineering and Systems, Escuela Superior de Ingeniería y Tecnología, Universidad de La Laguna, 38200 Santa Cruz de Tenerife, Spain

smartphone and tablet market revolution. From an architectural standpoint, modern SoCs are shared memory heterogeneous systems which integrate multicore CPUs and multiple accelerators, such as Graphics Processing Units (GPUs) and other specialized processors. Although the mobile landscape is very heterogeneous, we find that, in terms of operating systems, the vast majority of these devices run Android [17]. For that reason, our work focused on improving modern mobile applications performance targets that platform.

On Android, applications are mainly written on the Java programming language, but it is also available a Native Development Kit (NDK) that allows implementing parts of an application using C/C++ through the Java Native Interface (JNI). This is intended to allow reusing existing native libraries on Android applications, but it enables an application to make use of system libraries provided by vendors as well. This is the case of OpenCL drivers, not supported by Android, but provided by most of the main SoC vendors [5,11,13,18].

OpenCL is a high-performance development framework that allows the exploitation of many kinds of accelerators in heterogeneous platforms. It provides a runtime used to allocate and manage data transfers between processors, and to run parallel code on them. The standard defines OpenCL C as the programming language in which parallel functions, called kernels, are defined. However, it is still an open problem writing performance-portable OpenCL code. There is a need for auto-tuning methods to solve this problem, because of the high rate at which new SoCs reach the market and the cost of optimizing performance for each program and architecture.

Regarding auto-tuning, there are static approaches based on providing parameterized implementations to a system that, at compile time, explores different implementations and parameters until it finds the best tuning for that architecture in a certain amount of time. This is the case of ATLAS [19], a widely used implementation of BLAS. The Halide DSL [14] also follows this approach, but it applies it to image processing pipelines. It defines a functional DSL used to construct these pipelines, and it can stochastically search for ways to tradeoff between data locality, parallelism and redundant computation. One of the main shortcomings of this approach is that it is not well suited to the mobile development ecosystem, where programs get cross-compiled without knowledge of the target device. Other options are based on theoretical performance modeling [6,7,9], relying on short tests at installation time in order to estimate system parameters and recompile the tuned routines.

Tiling or cache-blocking optimizations, also called thread-coarsening on the SIMT[1] execution model, are a very common way of improving memory locality, having been successfully applied in many cases for optimizing code on multicore architectures [20]. More recently, their potential and difficulties for achieving OpenCL performance portability among multicore and GPU architectures have been discussed by several authors [1,12]. They focus on the differences between CPU and GPU code optimization [16,21], explore several different optimizations at once [8] or they focus on specific types of problem [10,15]. However, due to the current relevance of the mobile ecosystem, we believe the effect of tiling optimizations on performance portability on mobile GPU-accelerated codes should be studied. Mobile GPUs have

[1] Single Instruction Multiple Threads.

architectural differences with respect to their desktop counterparts, such as their unified system memory, the usage of hardware-managed caches and power constraints, and to the best of our knowledge there are currently no published works focused on the effects of tiling optimizations on these architectures.

The main contributions of this work are the following:

- We explore the fitness of tiling optimizations on mobile GPUs for performance
- We identify the variables that play a role on the selection of the optimal tile size
- We characterize the performance behavior of different kernels and devices depending on tile sizes and shapes
- A range of relevant stencil-based image processing and scientific codes is accelerated and analyzed
- We define a method for tiling GPU kernels considering memory coalescence, in a way susceptible of being implemented by automatic code generation tools
- A novel adaptive methodology for auto-tuning of tiling optimizations on mobile devices is designed and implemented, aiming at solving the performance portability problem with the smallest overhead possible

Our contributions help mobile developers understand the potential performance improvement that tiling optimizations can make on GPU-accelerated code, and devise a way in which it can be manually or automatically applied to any kernel, as well as providing an adaptive runtime system for the automatic parameterization of tiled code with a negligible overhead that is well suited to the mobile application lifecycle.

This paper is structured as follows: Sect. 2 describes how tiling optimizations can be efficiently implemented on mobile GPUs, Sect. 3 explains our methodology for dynamic tiling in mobile applications. In Sect. 4 a large set of experimental tests is presented and discussed, and Sect. 5 finishes with conclusions and future work.

2 Tiling optimizations on GPU

Tiling optimizations are a kind of loop optimization consisting on the division of a global iteration space into chunks, or tiles, that form smaller iteration sub-spaces. Processing each of these tiles separately can improve data locality because it favors the access of closer memory locations when the input size grows. Another use of this technique in GPGPU computing is to increase the granularity of a certain kernel, defining granularity as the amount of work each GPU thread carries out in parallel execution. By tuning the tile size it is possible to reach a compromise between parallelism and thread creation overhead. This is also referred to as thread-coarsening.

Figure 1 illustrates the order of execution of sequential and massively parallel implementations of some code running over a bidimensional range. Each cell represents the iteration number, or time step, of execution of each element within a range of size $M \times N$. Hence, multiple cells holding the same number are executed in parallel. The first case could represent a sequential code running on a CPU, in contrast to what the corresponding parallel GPU implementation would look like, assuming an embarrassingly parallel problem. Figure 2 shows the order of execution of sequen-

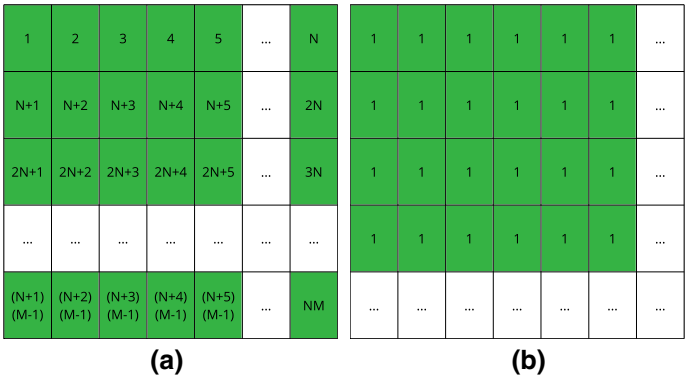

Fig. 1 Sequential and parallel execution models over 2D domain. **a** Sequential, **b** Parallel

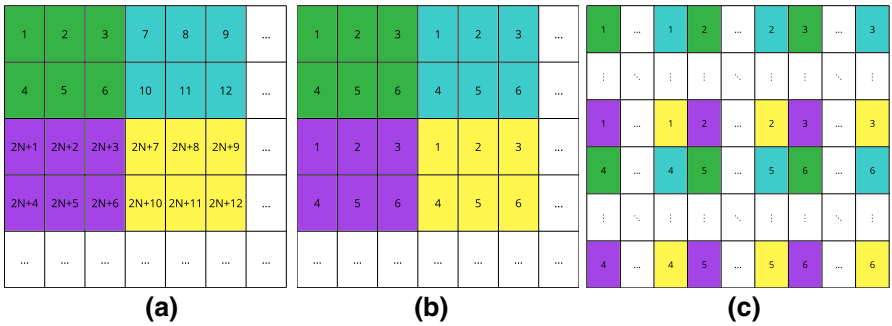

Fig. 2 Tiled (size 2x3) sequential and parallel execution models over 2D domain. **a** Sequential. **b** Parallel-Simple. **c** Parallel-Coalesced

tial and parallel tiled codes, using a 2x3 tile size and using colors to mark each tile. A tiled CPU code would look like shown in Fig. 2a, where the iteration order is clearly modified. Figure 2b shows how a straightforward implementation of tiling on GPUs would impose an execution order, reducing the number of threads running in parallel.

If we look closely at Fig. 2b, however, it is clear that parallel memory accesses corresponding to any given time step are scattered throughout the whole domain. This would correspond to uncoalesced memory accesses, which significantly hurt GPU performance. In order to avoid this problem, a coalesced tiled implementation has to be used instead, as shown in Fig. 2c. The skeleton for writing such kernel is detailed in Fig. 3. By considering memory coalescence, better performance increases are achievable using this simple code optimization on mobile and desktop GPUs. The code for doing coalesced tiling optimizations should be modified as follows:

– The size of the global range must be divided by the tile size of each dimension, using the mathematical ceiling. This adds padding threads to domains not multiple of the tile size.

```
1  void __kernel tiling2D (image2d_t srcPxs, image2d_t outPxs) {
2    int x = get_global_id(0), y = get_global_id(1);
3    int domainSizeX = get_image_width(outPxs), domainSizeY = get_image_height(outPxs);
4
5    for (; y < domainSizeY; y += get_global_size(1)) {
6      /* Backup primitive parameters modified in kernel... */
7
8      int __tmpx;
9      for (__tmpx = x; x < domainSizeX; x += get_global_size(0)) {
10       /* Regular kernel code... */
11     }
12     x = __tmpx;
13
14     /* Restore modified primitive parameters... */
15   }
16 }
```

Fig. 3 Structure of coalesced tiled kernel code working on OpenCL 2D images

– Inside the kernel, for each dimension of the range, the global size has to be calculated. This may be obtained from kernel arguments.
– The original kernel code must be surrounded by a skeleton as described in Fig. 3, which ensures a performant iteration order and coverage of the whole domain, while avoiding the need for using padded buffers.
– A primitive (i.e. `int`, `float`, ...) parameter passed into a kernel is expected to have the same initial value in each thread, and modifications to that parameter should always be local to the thread. However, since the same memory is reused for all iterations inside of a tile, it is necessary to make sure that modifications to such parameters are undone before continuing into the next iteration.
– Return statements inside the kernel have to be replaced by an increment on the innermost index variable and a `continue` statement, so that they do not prevent other iterations from being executed.

In [12] an alternative approach for automatically applying tiling optimizations to an OpenCL kernel is described. Their approach is able to avoid redundant computations within each thread through a divergence analysis, and the resulting code would be functionally equivalent to an unrolled version of our proposed transformation. However, they do not take into account the possibility of a thread performing local updates to primitive parameters, which should be reverted or replicated to maintain correctness. In addition, their approach assumes global domains to always be multiples of the tile size, which forces adding padding to memory buffers. This is not always the best option, since many kernels working on the same buffers could require different tile sizes for optimal performance. The required padding to fit various tiling configurations in these cases could increase significantly.

In general, the implementation of these optimizations is simple, but finding the optimal tile size is a very time-consuming task. The main problem is that the best tile size in each case depends on several parameters, such as the hardware architecture, the algorithm or the input size. Because of that, much experimentation has to be done prior to selecting a tile size. Many cases even display a negative impact on performance, so tile size selection is of prime importance.

3 Dynamic tiling on mobile devices

Due to the importance and difficulty of finding tile sizes that would provide performance gains in any given case, it is clear that an automated solution to this problem is needed [1]. Experimentation done in Sect. 4 shows the variables involved in finding the optimal tile size, so auto-tuning is deemed necessary for tiling optimizations.

One possible solution is to analytically determine the tile size parameter from data gathered at compile and runtime about all factors that impact performance. However, due to the high amount of interdependent variables and architectures, and the difficulty of representing relevant code features as parameters for an exact algorithm, this option becomes very difficult to implement [3,4].

The simplest option is to test for each algorithm and possible input, and in each architecture considered, the performance obtained with each tile size. After such benchmarking, the optimal tile size for each case would be found. Results could then be used to select at runtime the best tile size found depending on all variables considered during testing. If all relevant variables were analyzed during experimentation, the optimal tiled execution would be guaranteed. However, the time requirements of this approach and its low applicability to new parameters make it unprofitable.

Existing auto-tuning approaches such as that of ATLAS or Halide are limited in that they only take architecture and algorithm into account and that they require running a large set of benchmarks in the target device in advance. These approaches do not suit the mobile environment, in which applications with long startup times are disliked by users, and in which they may be started and stopped frequently.

Our solution to this problem is an adaptive method which iteratively explores the solution space of the execution time function depending on the tile size. We implement this as a native runtime system. Since it has been thought for use in mobile devices, we propose storing the current exploration status in the device, progressing as each kernel is repeatedly executed, and avoiding running any benchmarks in advance. Algorithm 1 shows how a kernel call is modified in order to use this system.

Algorithm 1 Dynamic tiling runtime interface usage

```
1: procedure AUTOTILINGKERNEL(tilingdb, range, kernel, params ...)
2:     exploration_state ← query_tiling(tilingdb,kernel,range)
3:     tile_size ← explore(exploration_state)
4:     adjusted_range ← ⌈range / tile_size⌉
5:     event ← kernel<adjusted_range>(tile_size,params...)
6:     wait(event)
7:     update(tilingdb,exploration_state,exectime(event))
8: end procedure
```

Every time a tiled parallel method is called, the dynamic tiling runtime is queried in order to obtain the next tile size to use. The execution time of the kernel is measured, and the runtime is updated with the results of that execution. In this way, this runtime is able to explore the solution space without severely degrading application performance. The more times a kernel runs, the closer the tile size gets to the optimal value, until it reaches a local minimum. Since each update is stored in persistent storage, the optimization progress is not lost across application runs.

Manual experimentation done in Sect. 4.2 on the tiling exploration space of several algorithms shows that, in addition to the algorithm and device, the input size is an important factor on selecting the optimal tile size. For that reason, the dynamic tiling runtime maintains independent explorations for each tiled method and input size. This is also carried out independently on each device, so we ensure the main variables impacting performance are taken into account.

The exploration function we selected for our dynamic tiling implementation is a greedy local search algorithm. Starting from a 2×2 tile size, it exponentially increases the tile size on each dimension until an improvement in performance over the best current tile is obtained. If no improvements are achieved, then the current best tile is used from that point onwards. If there is an improvement, the best tile is updated and the same procedure is repeated until convergence. This methodology will stop at the first local minimum found, from which no improvements can be obtained by increasing the tile size in one dimension. Although not optimal, this method achieves a negligible overhead and a very quick convergence, which we find it is usually not far from the best choice. Furthermore, no additional runs of the kernel need to be done in order to auto-tune it, because it adapts as the application uses it. Experimental evidence shows that the execution time function depending on tile size is irregular, but in some cases tends to resemble a parabolic shape. This may explain the good results obtained with such a simple exploration method.

With the goal of avoiding the exploration of bad tile sizes when some knowledge of the algorithm behavior has been gathered in the platform, current optimal tile sizes can be used as a starting point for explorations on new input sizes. When the tile size for a new input size on an algorithm for which there is already tiling data is requested, instead of starting the search from the beginning, the best tile size for the most similar input size is used. Our hypothesis is that, for any given kernel, optimal tile sizes for similar input sizes are also similar.

When the exploration process stops, if it started from the best tile size found for another input size, it is restarted in the opposite direction. Instead on iteratively increasing the tile size, it is decreased until no improvements are achieved. This is done so that, in these cases, smaller tile sizes are explored as well.

4 Computational results

4.1 Experimental setup

The two devices we used as testbed platform, represent two of the major SoC architectures that are most widely used in modern handheld devices, which are Qualcomm Snapdragon and Samsung Exynos. Therefore, the results we obtained are applicable to a large portion of devices currently in use.

– **Sony Xperia Z (labelled SXZ):** Based on a Qualcomm APQ8064 Snapdragon S4 Pro SoC with a Quad-core Krait CPU @ 1.5GHz and an Adreno 320 GPU with 4 OpenCL compute units and 2GB of shared RAM. Its GPU has 32KB of cache memory and 8KB of local memory.

– **Odroid-XU3 (labelled XU3):** Based on a Samsung Exynos 5422 Octa SoC with dual ARM CPUs (Cortex-A15 @ 2GHz and Cortex-A7 @ 1.3GHz) and a 6-core ARM Mali-T628 MP6 GPU with 2GB of shared RAM. Its GPU has 128KB of cache memory and 32KB of local memory.

We developed sequential Java, and regular and tiled OpenCL implementations of the 2D Gaussian Blur, Discrete Laplacian, Pattern Thinning, Heat and Poisson kernels [2]. They represent relevant algorithms found on regular mobile applications. Though they were repeated several times in order to increase accuracy, background services would often be killed and rebooted during testing, producing variations in performance. Thermal throttling had to be taken into account as well, since temperature rises running compute-intensive benchmarks reduces performance. For these reasons, we repeated each test depending on how long each took until thermal throttling started, and we added pauses to let the devices cool down. This helped reduce the overall experimental error.

4.2 Finding the optimal tile size

We have carried out extensive benchmarking in order to identify the parameters that impact performance when applying tiling optimizations on mobile GPUs. All of our benchmarks have been integrated with an Android Java application, through the Android NDK, so as to better represent a real use case. A wide range of tile sizes has been explored in order to visualize the impact on performance that they have in each case. Performance graphs in this section have been obtained by measuring execution times including all involved data movements and overheads. We define the normalized execution time as the time that each test takes relative to its un-tiled counterpart, averaged over a set of different input sizes.

Architectural differences are an important factor toward the optimal selection of a tile size, as shown in Fig. 4. Different algorithms also behave differently, as Fig. 5 demonstrates. Other less obvious features, such as the input size or the actual input data, respectively, studied in Figs. 6 and 7, prove to be relevant factors to consider as well. Failing to take into account any of these features when tuning a tiled kernel results in a significant performance penalty. Static or manual approaches cannot take all these features into consideration, so runtime systems are needed for this purpose.

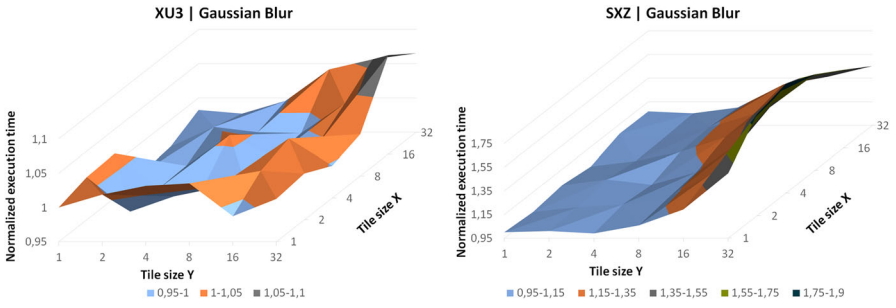

Fig. 4 Gaussian Blur tiling in different devices (best tiles $1x8$ and $1x2$)

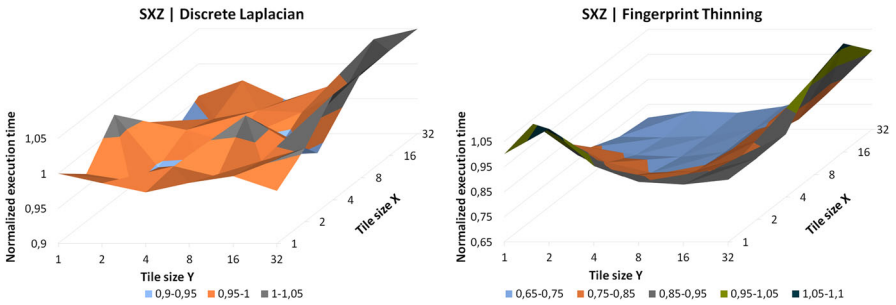

Fig. 5 Discrete Laplacian and Thinning tiling in Sony Xperia Z (best tiles 1*x*16 and 1*x*32)

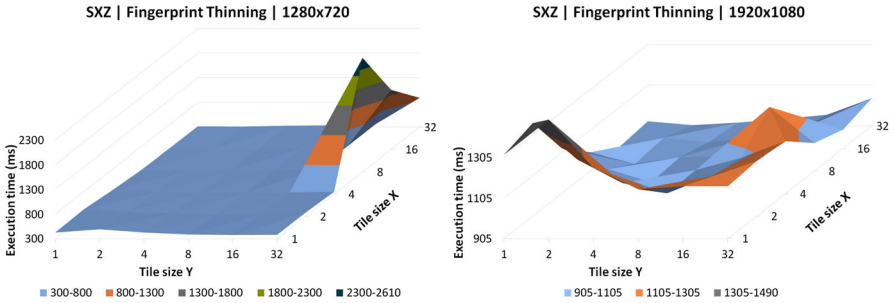

Fig. 6 Thinning tiling in Sony Xperia Z (best tiles 1*x*1 and 1*x*16)

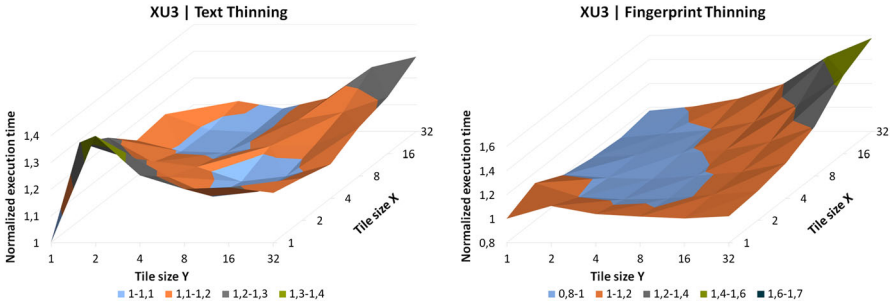

Fig. 7 Thinning different images with tiling in Odroid-XU3 (best tiles 1*x*1 and 1*x*16)

Algorithm differences aside, which impact performance due to differences in computation with memory access ratios and memory access patterns, it is interesting to notice that Fig. 6 does not show similar graph shapes for the same algorithm running on the same device, by only varying the input size. Also, the effect of branching on performance is of vital importance particularly on GPUs, whose performance depends on avoiding branching within thread groups and accessing memory in contiguous blocks. If either accessed memory addresses or conditions for branching depend on input data, that is also a very relevant factor for efficiency, as Fig. 7 shows. This is, however, a much more difficult feature to quickly and automatically extract from the inputs than their size.

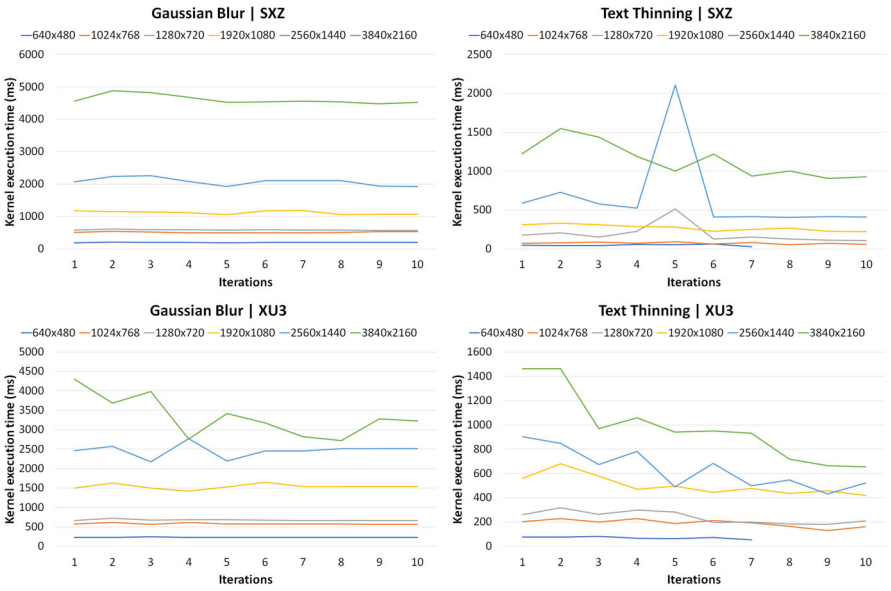

Fig. 8 Dynamic tiling exploration for the Gaussian Blur and Thinning algorithms

4.3 Dynamic tiling results

Results shown in Sect. 4.2 demonstrate that many variables come into play when selecting the optimal tile size and that it is impossible to find a single pattern which allowed developers to obtain the optimal tile size in advance. Tiling optimizations can, in some cases, allow up to a 50% increase in kernel execution performance. However, a bad tile size selection can entail a significant performance penalty. The solution is to tune the tile size for every device, algorithm, input size and processor.

Solving this problem by hand is impractical, so we propose an automatic approach to do so. Our implementation of automatic dynamic tiling is able to take into account most of the parameters impacting tiled execution performance. Performance improves over time until convergence is reached. Figure 8 shows how kernel execution times evolve as the number of executions increases.

In certain cases, the performance stays stable from the beginning, such as the Gaussian Blur algorithm on the SXZ. This may indicate a flat exploration space or starting at a local minimum. In other cases, such as Thinning a text image in the SXZ, we encounter very high peaks significantly reducing performance for a single iteration. On average these do not significantly impact application performance, since such parameters are only explored once at most.

Oftentimes, the described exploration methodology converges very fast. In our testing, we found that in 10 iterations or less, the final tile size had usually been already found, which is good for performance stability but may mean that the global optimum is never reached. Parabolic-shaped search spaces, which we have encountered in some of our tests, do not present this problem. We evaluate the relative execution time obtained

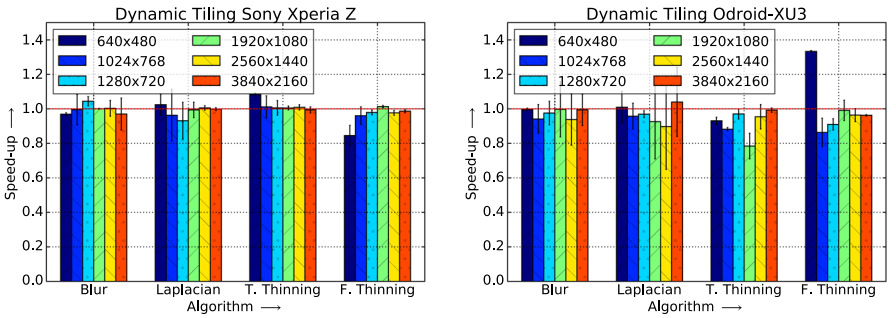

Fig. 9 Relative performance of dynamically selected tile sizes over the average best

with the tile size found by our dynamic approach, in comparison with the best average tile size in Fig. 9. The best average tile size is determined for each combination of algorithm and device, and it is selected as the best performing in average across all considered input sizes. This is intended to represent a kernel hand-tuned for quick execution on any input size.

These results show that dynamic tiling performance is usually within a 90% of the best performance obtained by any tiling optimization of the kernel. However, there are also cases where the performance falls to quite worse values. Several instances in which dynamic tiling achieved a better result than statically selecting the average best tile size can be found as well.

Our dynamic tiling methodology does not currently allow the selection of a smaller than 2x2 tile size, which explains the punctual bigger gaps to optimal performance. In many cases, not tiling is the best option, as we see in Sect. 4.4. However, because the dynamic tiling methodology maintains a different tile size for each device, algorithm and input size, it can obtain better results than a static selection of the tile size according to the best average tile size for each algorithm and device. That added granularity explains greater than 100% relative performance cases.

4.4 Final results

We compare the performance of a sequential Java implementation to a manually tiled, a dynamically tiled and a baseline OpenCL implementation. Tiled implementations share the same kernel code, which is parameterized so it can use any tile size. While dynamic tiles are selected at runtime according to the methodology described in Sect. 3, manual tiling statically chooses for each algorithm and device the best performing tile size over a set of input sizes. All benchmarks in this section show the speedup over the reference Java implementation. Each graph represents one of the tested algorithms, and each of them contains a group of bars representing the input sizes used. Non-tiled OpenCL variants are labelled as *OCL*, and the manual and dynamic tiled variants are, respectively, named *M. OCL. T* and *D. OCL T*.

From the results shown in Fig. 10 we find that in problems with higher computation density, like Pattern Thinning, we achieved significative improvements through tiling. There are cases where dynamic tiling does not provide these big improvements

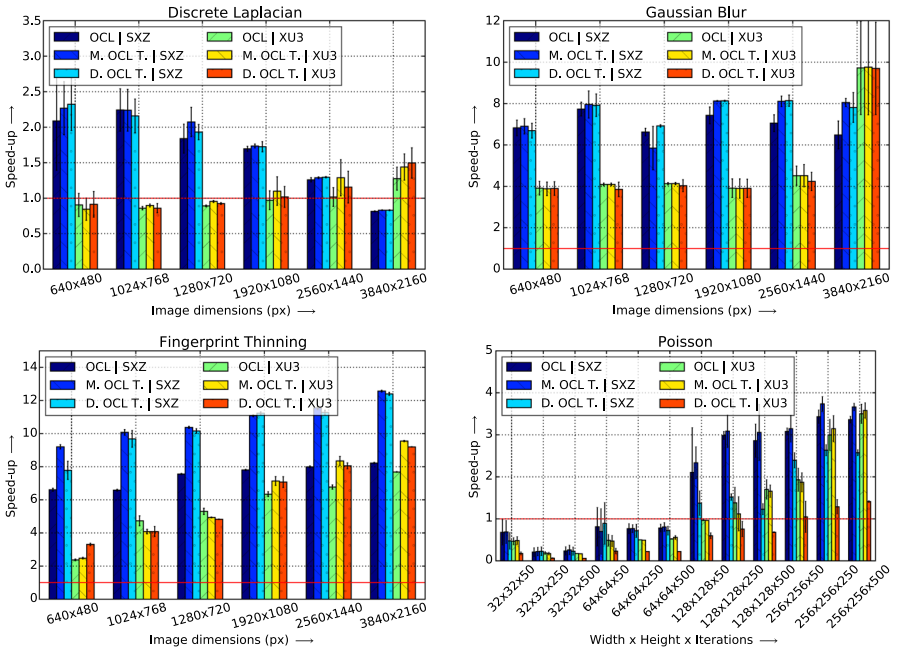

Fig. 10 Speedups of non-tiled (OCL), manual (M. OCL T.) and dynamic (D. OCL T.) tiled OpenCL implementations over equivalent Java code (higher is better)

over its non-tiled counterpart, as well. The Discrete Laplacian and Gaussian Blur are such examples, though in these cases manual tiling also did not improve performance greatly. In the case of very quick kernels that have to run many times, like Heat and Poisson, our dynamic tiling implementation tends to drastically reduce performance. This is because it requires the execution of a kernel to finalize before another one can be queued for execution. Algorithms that iteratively call a parallel kernel suffer from low occupancy due to this serialization. Its solution would be the asynchronous processing of kernel execution times and update of the exploration status.

5 Conclusion and future work

We propose a methodology for tiling optimizations on OpenCL code for mobile GPUs, having memory coalescence in mind to improve performance. It is also simple to integrate in an automatic code generation tool for completely transparent optimization, because it is kernel-agnostic. Benchmarks carried out in modern SoCs show that we can obtain reasonable performance improvements through this optimization.

Our dynamic tiling methodology reduces the effort required to use this feature in a performance portable way, but more work is required in order to guarantee that it does not suppose a performance penalty for situations when not tiling is the best option or for kernels running for short amounts of time repeatedly. In particular, allowing the

asynchronous update of the dynamic tiling exploration data and implementing a more advanced search strategy are possible ways to improve this system.

There are other code optimizations that can be done to GPU codes in mobile devices that we have not explored in this work. Factors like workgroup or block size can affect performance due to a different mapping of threads to hardware compute units and memory access patterns. Our dynamic tiling runtime system can be redesigned in order to help auto-tuning these other optimizations.

Performance measurement in mobile devices is especially hard due to thermal throttling issues and lack of control over background services. We find that there is a very significant variation over consecutive runs of the same code. In order to improve the accuracy of results, we need to pinpoint all sources of uncertainty and come up with solutions for them.

References

1. Acosta A, Almeida F (2015) Towards the optimal execution of renderscript applications in android devices. Simul Model Pract Theory 58:55–64. https://doi.org/10.1016/j.simpat.2015.05.006
2. Afonso S, Acosta A, Almeida F (2017) Automatic acceleration of stencil codes in android devices, pp. 81–95. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-65482-9_6
3. Almeida F, Andonov R, González D, Moreno LM, Poirriez V, Rodríguez C (2002) Optimal tiling for the RNA base pairing problem. In: SPAA, pp. 173–182. https://doi.org/10.1145/564870.564901
4. Andonov R, Rajopadhye S (1997) Optimal orthogonal tiling of 2-d iterations. J Parallel Distrib Comput 45(2):159–165. https://doi.org/10.1006/jpdc.1997.1371
5. ARM: Mali graphics and multimedia processors. https://developer.arm.com/products/graphics-and-multimedia/mali-gpus
6. Boratto M, Alonso P, Giménez D, Barreto M (2013) Oliveira K Auto-tuning methodology to represent landform attributes on multicore and multi-gpu systems. In: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '13, pp. 125–132. ACM, New York, NY, USA. https://doi.org/10.1145/2442992.2443006
7. Boratto M, Alonso P, Giménez D, Lastovetsky A (2017) Automatic tuning to performance modelling of matrix polynomials on multicore and multi-gpu systems. J Supercomput 73(1):227–239. https://doi.org/10.1007/s11227-016-1694-y
8. Chu SL, Hsiao CC (2013) Methods for optimizing opencl applications on heterogeneous multicore architectures. Appl Math Inf Sci 7(6):2549
9. García LP, Cuenca J, Giménez D (2007) Including improvement of the execution time in a software architecture of libraries with self-optimisation. In: ICSOFT (SE), pp. 156–161. Citeseer
10. Holewinski J, Pouchet LN, Sadayappan P (2012) High-performance code generation for stencil computations on gpu architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing, pp. 311–320. ACM
11. Imagination: A quick guide to writing OpenCL kernels for PowerVR Rogue GPUs. https://www.imgtec.com/blog/a-quick-guide-to-writing-opencl-kernels-for-rogue/. Accessed 9 Oct 2018
12. Magni A, Dubach C, O'Boyle MFP (2013) A large-scale cross-architecture evaluation of thread-coarsening. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 11:1–11:11. ACM, New York, NY, USA. https://doi.org/10.1145/2503210.2503268
13. Qualcomm: Adreno GPU SDK. https://developer.qualcomm.com/software/adreno-gpu-sdk. Accessed 9 Oct 2018
14. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. 48(6):519–530. https://doi.org/10.1145/2499370.2462176
15. Rocha RCO, Pereira AD, Ramos L, Góes LFW (2017) Toast: automatic tiling for iterative stencil computations on gpus. Concurr Comput Pract Exp 29(8):4053. https://doi.org/10.1002/cpe.4053

16. Shen J, Fang J, Sips H, Varbanescu AL (2013) Performance traps in opencl for cpus. In: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 38–45. IEEE
17. StatCounter: Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide/2017. Accessed 9 Oct 2018
18. Vivante: Vivante Vega GPGPU technology. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html. Accessed 9 Oct 2018
19. Whaley RC, Petitet A, Dongarra JJ (2001) Automated empirical optimizations of software and the atlas project. Parallel Comput 27(1):3–35. https://doi.org/10.1016/S0167-8191(00)00087-9
20. Wolfe M (1989) More iteration space tiling. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing '89, pp. 655–664. ACM, New York, NY, USA. https://doi.org/10.1145/76263.76337
21. Zhang Y, Sinclair M, Chien AA (2013) Improving performance portability in opencl programs. In: ISC, pp. 136–150. Springer