Exploiting Kepler Capabilities on Zernike Moments

Antonio Ruiz, Manuel Ujaldón
Computer Architecture Department
University of Malaga, Spain
E-mail: {antruiz,ujaldon}@uma.es

***Abstract*—This work analyzes the most advanced features of the Kepler GPU by Nvidia, mainly dynamic parallelism for launching kernels internally from the GPU and thread scheduling via Hyper-Q. We illustrate several ways to exploit those features from a code which computes Zernike moments, using two different formulations: direct and iterative. This way, we compare how well they can deploy parallelism on the new generation of GPUs. The direct alternative tries to maximize parallelism, while the iterative one increases the operational intensity by reusing results coming from previous iterations. This has allowed us to increase the speed-up factor attained on Fermi architectures versus a code written in C and executed on a multicore CPU. We also succeed on identifying the critical workload which is required by a code to improve its execution on the new GPU platforms endowed with six more times computational cores, and quantify the overhead introduced by the new dynamic programming mechanisms in CUDA.**

I. INTRODUCTION

General purpose computing on GPUs (GPGPU) started a decade ago and since then it has transformed the High Performance Computing (HPC) arena with extraordinary acceleration factors. GPUs, designed with thousands of small but efficient cores, allow to deploy parallelism at multiple layers, helping CPUs to process those parts of an application which are more demanding on computational time and/or allow to benefit from massive data parallelism.

Disruptive programming models like CUDA or OpenCL have made GPUs popular to programmers of a very diverse background. However, it is still necessary to know at a basic level the new programming paradigm of these processors to be able to redesign applications and, at a more advanced level, if we pretend to exploit the whole set of enhancements introduced on a new architecture.

This work analyzes the Kepler architecture recently introduced by Nvidia, focusing on two pillars of its SMX multiprocessors: dynamic parallelism and Hyper-Q. The benchmark we have selected for this purpose is an algorithm which computes Zernike moments for characterizing images on very diverse scientific areas like biomedicine, robotics or topography. This algorithm was also studied for its efficient execution on Fermi, the previous GPU generation by Nvidia, which will be used here as departure point and reference for performance discussions.

Moment functions are integrals typically approximated by discrete sums when applied to pixelated images. They can be interpreted as a convolution of the image with a mask. However, moments are more attractive than convolutions because some are invariant to image translation, scale change and rotation [1], [2], [3]. Derived from the general concept, moments with orthogonal basis functions such as Legendre and Zernike polynomials can be used to represent the image by a set of mutually independent descriptors, with a minimal amount of information redundancy [4], [5]. The orthogonality property enables the separation of the individual contribution of each order moment, enabling its role as image descriptors.

Even though the compatibility with applications developed in CUDA is guaranteed in enhanced versions of the hardware, performance can be greatly improved if the code is optimized on a particular architecture, which requires a deep knowledge of software mechanisms for taking advantage of those advanced features. Unfortunately, not all the architectural changes are revealed by hardware vendors. Our contribution here goes to help programmers in this daunting task, connecting software elements with hardware capabilities, and discussing performance for a broad set of features.

This paper is structured as follows. Section II describes Zernike moments and the state of the art for its computational implementation. Section III presents the CUDA programming model, highlighting those mechanisms widely used along this work. Section IV provides details of the Kepler architecture and its main differences with respect to the previous Fermi GPU. Section V shows the departure point for our implementation of Zernike moments, introducing its peculiarities for exploiting GPU parallelism. Section VI describes strategies to follow for improving performance on the Kepler architecture. Experimental results and conclusions drawn from this work are outlined in the two closing sections.

II. ZERNIKE MOMENTS

A. Mathematical formulation

Zernike functions are a set of complex orthogonal functions with a simple rotational property which forms a complete orthogonal basis over the class of square integrable functions. The set of orthogonal Zernike moments for an image represented by the intensity of its pixels $f(r,\theta)$ with an order p and repetition q are defined as follows [4]:

$$Z_{pq} = \frac{p+1}{\pi} \int_0^{2\pi} \int_0^1 f(r,\theta) V_{pq}^*(r,\theta), r dr d\theta, \qquad (1)$$

where $V_{pq}^*(r,\theta)$ are the complex conjugates of the Zernike polynomials $V_{pq}(r,\theta)$ whose representation describe the unit disk defined as

$$V_{pq}(r,\theta) = R_{pq}(r) e^{jq\theta} \qquad (2)$$

with p an integer fulfilling

$$\begin{aligned} p \geq 0, 0 \leq |q| \leq p, p - |q| = par, j = \sqrt{-1} \\ \theta = \tan^{-1}(y/x), 0 \leq \theta \leq 2\pi \end{aligned} \tag{3}$$

Radial polinomials $R_{pq}(r)$ are defined as

$$R_{pq}(r) = \sum_{k=0}^{(p-|q|)/2} (-1)^k \frac{(p-k)!}{k!(\frac{p+|q|}{2} - k)!(\frac{p-|q|}{2} - k)!} r^{p-2k} \tag{4}$$

When we change the previous formulation from the continuous domain and polar coordinates to the discrete domain and cartesian coordinates, the equation 1 for an image function $f(x, y)$ of size NxN takes the following form

$$Z_{pq} = \frac{p+1}{\gamma N} \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} f(x_i, y_k) V^*_{pq}(x_i, y_k) \Delta x_i \Delta y_k \tag{5}$$

where $x_i^2 + y_i^2 \leq 1$ and γN is a normalization component which corresponds to the number of pixels within the unit disk whose coordinates are represented by

$$x_i = \frac{2i + 1 - N}{N}, \quad y_k = \frac{2k + 1 - N}{N} \tag{6}$$

When we add to the $f(x, y)$ image a rotation with an angle α, the Zernike moment V'_{pq} of an image is

$$V'_{pq} = V_{pq} e^{-jq\alpha} \tag{7}$$

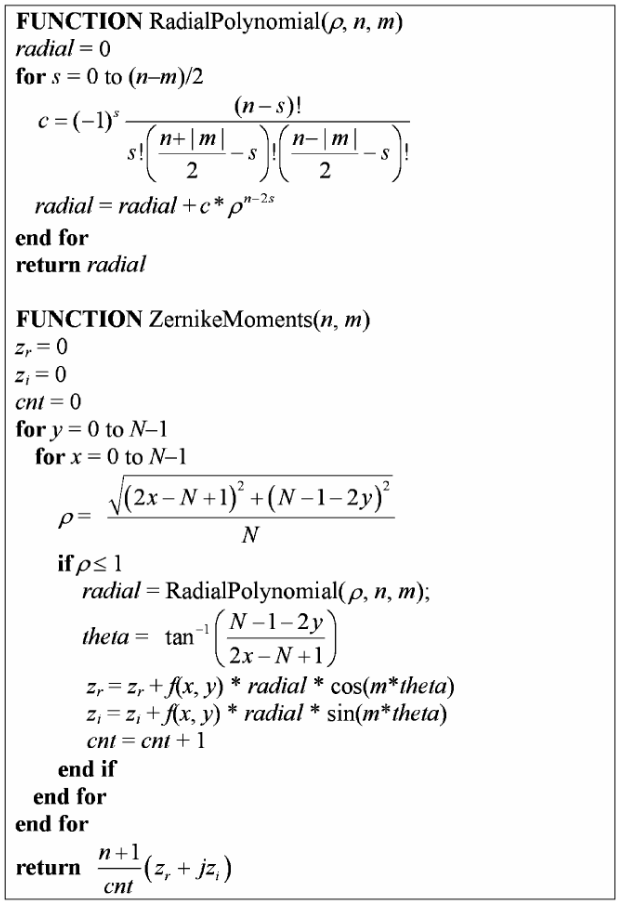
Since the rotation only modifies the phase of the Zernike moments, the absolute value is invariant to the rotation. The same happens when images are modified to be centered, scaled or changed through linear transformations. These features make them more suitable for image analysis than other existing approaches like Legendre moments [6], though at the expense of a higher computational cost [7].

B. Computational techniques

Figure 1 outlines the algorithm for computing Zernike moments of an order n and a repetition m following the equation 5, and assuming an NxN image size. Real and imaginary parts are expressed as z_r y z_i, respectively.

The computation of Zernike moments has been improved over the years [8]. From a computational perspective, we may distinguish two basic approaches:

1) **Direct methods:** Under this formulation, moments are computed individually without relying on others, basically applying the formula given by eq. 5. This way, they may constitute features characterizing an image after a selection of the most discriminant coefficients. This situation arises more often on image classifiers and image segmentation [9], [10].
2) **Recursive methods:** Here, the computation of a single order moment and an isolated repetition within an order cannot be computed without sweeping over all previous repetitions and orders. This allows to partially amortize the cost of previous computations, but it is only suitable in certain application domains like image reconstruction, where all moments (repetitions) are required up to a given order.
Recursive formulations have gained attention over the past ten years. For example, Chong et al. [11] propose a recursive approach and perform a survey comparing all alternatives, and Hwang et al. introduce novel ideas to speed up the process by exploiting the symmetry in the polar coordinates space [12].



```
FUNCTION RadialPolynomial($\rho$, $n$, $m$)
radial = 0
for $s$ = 0 to $(n-m)/2$
  $c = (-1)^s \dfrac{(n-s)!}{s!\left(\frac{n+|m|}{2}-s\right)!\left(\frac{n-|m|}{2}-s\right)!}$
  $radial = radial + c * \rho^{n-2s}$
end for
return radial

FUNCTION ZernikeMoments($n$, $m$)
$z_r = 0$
$z_i = 0$
$cnt = 0$
for $y$ = 0 to $N-1$
  for $x$ = 0 to $N-1$
    $\rho = \dfrac{\sqrt{(2x-N+1)^2 + (N-1-2y)^2}}{N}$
    if $\rho \leq 1$
      radial = RadialPolynomial($\rho$, $n$, $m$);
      $theta = \tan^{-1}\left(\dfrac{N-1-2y}{2x-N+1}\right)$
      $z_r = z_r + f(x, y) * radial * \cos(m*theta)$
      $z_i = z_i + f(x, y) * radial * \sin(m*theta)$
      $cnt = cnt + 1$
    end if
  end for
end for
return $\dfrac{n+1}{cnt}(z_r + jz_i)$
```

Fig. 1. A basic algorithm for the computation of Zernike moments.

III. The GPU Programming Model

This section introduces the elements of CUDA (*Computer Unified Device Architecture*) [13], which allows to deploy massive parallelism on the GPU at different levels.

The GPU hardware is structured on multiprocessors, endowed with SIMT (*Single Instruction Multiple Threads*) computational cores which share the control unit and a small but extremely fast shared memory. DRAM video memory is called global memory and it is accessible to all multiprocessors, at a slower latency but extraordinary bandwidth.

The CUDA programming model defines the following basic elements:

- **Threads**: The basic execution unit that is mapped to a single GPU core.
- **Blocks**: Batches of threads assigned to a single multiprocessor which share all the resources included in this multiprocessor, such as register file and shared memory. Block are scheduled on GPU multiprocessors, and executed via **warps**, the minimal working and scheduling unit. Until now, the warp size has always been 32 threads.
- **Grid**: Set of twin blocks in which the execution of a CUDA kernel can be decomposed to express parallelism.
- **Kernel**: Code excerpt delimiting a function to be ported to the GPU. It is executed by all threads defined in the grid of blocks, each of them working on a different data region thanks to its `blockID` and `threadID`, unique identifiers which are assigned to threads at run-time.
- **Stream**: Execution flow in parallel with other streams defined. When kernels are not associated to streams, they must follow a serial execution, that is, there can only be a single kernel in execution at a given instant, which takes all GPU resources. Using streams we can parallelize the execution of as many kernels as streams may exist, for a concurrent run on all multiprocessors available.

With all these elements, programmer must explicitly declare the number of blocks and the block size required for executing the kernel on the GPU. Large-scale applications deploying massive parallelism declare a huge number of blocks, as the block size cannot exceed 1024 threads.

IV. The Kepler architecture

Since CUDA inception in 2006, GPUs have evolved very fast and Nvidia has developed three hardware generations: Tesla (2008), Fermi (2010) and Kepler (2012)[14]. This work is focused on the latter and compared to its predecessor, Fermi.

Fermi extended the number of GPU cores up to 512, and double precision floating-point units up to 256. L1 and L2 caches were also introduced, with the L1 configurable in size along with the shared memory. There were also enhancements in context switches, atomic operations and ECC memory.

Kepler introduced the SMX multiprocessor, endowed with 192 cores for integer and single precision floating-point arithmetic, and 64 cores for double precision floating-point. There were initial versions with 13 and 14 SMXs called K20 and K20X, respectively. In November 2013, the K40 was released with 15 SMXs for a total of 2880 cores. SMX incorporates two major features: Dynamic parallelism and Hyper-Q scheduling [1], which represent the main target of our study in this paper. Before we describe those features, Table I performs a comparison between Fermi and Kepler based on those parameters which are more relevant to CUDA performance.

Regular applications and massively parallel code can benefit from higher number of *warps* and blocks which can simultaneously be executed on newer SMXs. Irregular and recursive algorithms rely more on dynamic parallelism and Hyper-Q, though at the expense of programmer's effort.

[1]A preliminar Kepler GPU named K10 did not include any of these two features.

TABLE I
Fermi and Kepler GPU families summarized.

GPU generation	Fermi	Kepler
Hardware model	GF100	GK110
Threads per warp	32	32
Maximum number of warps per multiprocessor	48	64
Active blocks per multiprocessor	8	16
Maximum block size (in threads)	1024	1024
Maximum number of threads per multiprocessor	1536	2048
Maximum number of registers per thread	63	255
Maximum dimension for the grid of blocks	$2^{16}-1$	$2^{32}-1$
Dynamic parallelism	No	Yes
Hyper-Q	No	Yes

A. Dynamic parallelism

On a hybrid CPU-GPU system, the efficient execution of applications with high degree of parallelism depends mostly on the versatility for distributing the work between them by exploiting the idiosyncrasies of each platform. Until 2013, a CUDA-enabled GPU was seen as a coprocessor helping the CPU with high speed-up factors, but low autonomy. Dynamic parallelism allows the GPU to launch its own kernels, create the events and threads required to control dependencies, synchronize the results and control the task scheduling. This frees the CPU, which can focus now on its own tasks in a more efficient manner. The GPU also contributes with a more straight processing of nested loops and recursive algorithms, and in general, benefits from a more natural computation of dynamic code and irregular data structures.

For example, now it is possible to determine at compile time the number of threads dedicated to process the new kernels created, and we can establish an initial set up with a more conservative parallelism deployment which avoids unnecessary computations on plain regions of an image, and increase parallelism at run-time over those areas which we see that require a more demanding processing as the computation evolves.

B. Hyper-Q

The search of an optimal scheduler to manage the GPU workload on a multiple stream code is one of the toughest challenges for its architecture. Fermi allows the concurrent execution of up to 16 streams, but they are implemented underneath using a single queue, which forces to serialize the execution. We can relax this constraint by reordering the kernels of each stream, but this process is much tougher on complex applications. Hyper-Q enables up to 32 concurrent queues between the CPU and the CUDA workload distributor on the GPU, endowing the design with flexibility and a performance leap without changes in the implementation. Now, each stream is managed independently on a different hardware queue, without affecting dependencies in neighbor queues, and streams may proceed in parallel coming from the same or other CUDA program, MPI process or POSIX thread (widely known as *p-thread*).

V. Implementation of Zernike moments

When optimizing Zernike moments for the Kepler architecture, our departure point will be the more recent version developed for GPU [9]. Moreover, in order to unify comparisons with those versions already existing in the bibliography, we will distinguish between direct and recursive methods.

Our baseline implementation belongs to the direct alternative for being more suitable to GPUs (skips data dependencies) and more efficient on applications where selected orders and repetitions have to be computed (the complete collection of orders and repetitions within each order is not required). Moreover, its implementation can take advantage of a couple of optimizations:

- **Parallelism.** Our algorithm applies the same set of operations on each pixel in an independent manner, thus allowing a more natural use of data parallelism.
- **Symmetry.** One of the heaviest stages for computing Zernike moments is the trigonometric calculations for the unit disk which are applied for the transformation from polar to cartesian coordinates. Symmetries on quadrants and even octants allow to reuse many computations [12], leading to speed-up factors of up to 8x on the GPU.

Our GPU implementation consists of five kernels which accept a grayscale image as input and return the Zernike moments as output relying on direct methods. If we adapt this process to the CUDA programming model, we have:

1) **Trigonometry for the unit disk.** The first stage processes the cartesian coordinates space with its trigonometric values (sin and cos) and distance, also discriminating those pixels which are outside the unit disk. This stage takes advantage of aforementioned symmetries.
2) **Zernike polynomials.** Given the distances and sin/cos values coming from the previous stage, we calculate the Zernike polynomials for each pixel. The sum given by equation 5 is performed on a shared memory space to avoid repeated accesses to global memory in CUDA.
3) **Application to the input image.** The result of the space obtained in the previous stage is multiplied with the input image.
4) **Pixel sum.** The sum of pixels within the unit disk is accumulated by using a sum reduction strategy. This type of operator has been deeply studied for its CUDA implementation, and we can luckily amortize this effort here.
5) **Pixel components sum.** A similar operation is performed for each pixel component to add its contribution to the Zernike moment, and again, a reduction operator is implemented following strategies already established.

Once the baseline implementation has been roughly described, Table II outlines the hardware we have used along our experimental study. We count on two GPUs of different generations: A GF100 Fermi for a reference with the previous architecture, and a Kepler GK110 which allows to quantify improvements attained on newer SMXs with dynamic parallelism and Hyper-Q.

VI. Optimizing Zernike moments on Kepler

This section analyzes those parts of our GPU implementation which can potentially benefit from features introduced in the Kepler architecture, though we can anticipate that not everyone we subscribe is going to lead to a faster execution.

A. Recursivity

We start describing the implementation of the recursive method on the GPU, despite it looks like more challenging that the direct counterpart for an efficient execution [9].

Within recursive methods, *q-recursive* is the more recent and efficient [11] to compute all repetitions of Zernike moments which correspond to a given order. The first couple of moments belong to the two highest repetitions, and from that departure point, lower repetitions are progressively computed through static expressions involving those two moments immediately higher. This methodology is successful on the CPU whenever we compute several moments for the same order, but some transformations are required on the baseline implementation as it was already described in section V.

The CUDA code for this method is going to be built by following an iterative process that calculates the Zernike polynomials on each step for a given repetition taking as input those moments previously stored. The memory space increases proportionally to the number of repetitions to be computed, but the complexity for this algorithm is lower and kernels which do not depend on the repetition can be amortized for all the iterations to be performed.

Kernels numbered as 1 and 4 in section V remain intact, while all the others require to perform the following changes:

- Kernel 2, which applies Zernike polynomials to every pixel must be split in two, as the nature of the iterative algorithm prevents from linking the processing of Zernike polynomials with its application to the point of the cartesian coordinates space. Now, we have:
  - 2.1 Zernike polynomials in its recursive way. It processes Zernike polynomials recursively. This kernel is executed as many times as repetitions we have for the specific order(s) of the Zernike moments to be computed. Each processed value uses its own memory space.
  - 2.2 Application to the cartesian space. Zernike polynomials are applied over the whole space, together with the trigonometric functions preceding them.
- Kernels 2.1, 3 y 5 increase its workload the same factor than the number of repetitions. This extra work is performed by each thread, which distinguishes uniquely the partition of the memory space where it has to access thanks to its block ID and thread ID within the block.

B. Dynamic parallelism

Dynamic parallelism can be applied in several ways to Zernike moments depending on the workload we offload from the CPU to the GPU. We now describe different strategies that can be eventually combined and their contributions:

1) **Launch kernels from the GPU.** The calls to the five CUDA kernels deployed in the GPU version of the direct method are shifted to the GPU scope, and now we launch initially a single kernel composed of a single thread and block. This root kernel performs all subsequent calls to the code for computing Zernike moments on the GPU, following what it was done in the original version implemented on the CPU.
2) **Calculate a repetition from each thread.** In this case, we exploit dynamic parallelism to calculate all moments for a given order. In the original version following the direct method, it would be necessary to apply a loop for iterating on the number of repetitions. By applying dynamic parallelism, the implementation would be equivalent to that of the previous paragraph, but with each kernel having one thread for each repetition. In order to avoid redundancies in the computations, those kernels in common for all the repetitions would be processed from the same thread.
3) **Parallelize the loop of the Zernike polynomials.** The `for` loop required to calculate Zernike polynomials using the direct method (see Figure 1) is a good potential candidate to take advantage of dynamic parallelism. Each pixel requires this calculation, so each of them will launch a new kernel to process concurrently the work of such loop.

C. Hyper-Q

In order to exploit Hyper-Q, we have to set up a process decomposed in several flows of concurrent execution which are free of dependencies. This goal is attainable on Zernike moments, similarly to the previous section where we needed to calculate all the repetitions for the Zernike moments of an specific order.

Hyper-Q is transparent to the programmer. All we have to do is enable CUDA streams to benefit from multiple queues of concurrent execution. The increment in the number of queues from 16 in Fermi to 32 in Kepler becomes crucial in the new architecture given the larger number of CUDA cores, because now, with a number close to three thousand, it is more likely that a single kernel can only occupy a fraction of them. By distributing kernels in different CUDA streams whenever possible, additional streams may use those processors that are idle after the first grid of blocks is deployed for the execution of a first kernel.

VII. Experimental results

Table II summarizes hardware features for the GPUs we have used along our experimental evaluation.

We have used a single precision floating-point data type and all image sizes starting from 64x64 until 2Kx2K pixels. The maximum order for the Zernike moments has been 34 because it is a limit imposed by the hardware for the calculation of the factorial numbers which are used within the Zernike polynomials.

TABLE II
Summary of GPU features for the hardware we have used to run our implementations.

GPU generation	Fermi	Kepler
Commercial model	Tesla C2075	Tesla K20c
Type of multiprocessor	SM (32 cores)	SMX (192 cores)
Number of multiprocessors	14	13
Total number of CUDA cores	448	2496
Frequency for the cores	1.15 GHz	710 MHz
Peak performance	1030 GFLOPS	3.52 TFLOPS
Frequency for the memory	2x 1566 MHz	2x 2600 MHz
Bus width for the memory	384 bits	320 bits
Memory bandwidth	148 GB/s.	208 GB/s.
Memory size (GDDR5)	6 Gbytes	5 Gbytes
Bus from/to CPU	PCI-e x16 2.0	PCI-e x16 2.0
ALUs / SM(x)	32	192
32-bit FPUs / SM(x)	32	192
64-bit FPUs / SM(x)	16	64
Load/Store Units / SM(x)	16	32
Special Function Units / SM(x)	4	32

A. Architectural changes: SMX

First of all, we pretend to quantify those improvements due to architectural changes, without making any change on the CUDA implementation. Table III shows the execution time on Fermi and Kepler for our baseline implementation described in section V, and compares speed-up factors. Those factors move in a window between 0.67x (slower) and 2.64x. The minimum value represents an execution time 1.5 times slower, which corresponds to the smaller image size, and similarly, the maximum speed-up of 2.64x is attained when the workload reaches its maximum value.

The GPU as general-purpose coprocessor, along with its data-parallel model, shine more when the input image size grows, which explains why Kepler migration runs faster when the number of pixels to process becomes huge. The smaller image size is 4096 pixels, evenly distributed among CUDA blocks spread through GPU multiprocessors. Fermi's SM multiprocessor may process up to 2 *warps* concurrently, which leads to 896 pixels to process in our Fermi GPU (2x32x14). Kepler's SMX multiprocessor reaches up to 8 *warps* for a total of 3328 simultaneous pixels to be exploited by hardware resources underneath (the GPU back-end). In this particular case, even though the architecture has been improved, resources are underused, and the much higher frequency in Fermi, leads to a winner execution.

With a size of 128x128 pixels, the workload saturates the maximum number of threads that Fermi can process concurrently, whereas Kepler reaches 75.7% of its maximum occupancy. For this image size, performance is similar on both architectures because Kepler does not exploit all hardware capabilities but can issue and schedule warps all together (Fermi requires a second round to schedule remaining blocks).

For images of larger size, Kepler improvements grow until they reach 100% of hardware resources available being used.

TABLE III
EXECUTION TIMES (IN MILLISECONDS) TO PROCESS ALL REPETITIONS FOR A GIVEN ORDER FOR ZERNIKE MOMENTS THROUGH DIRECT METHODS. INPUT IMAGES ARE SQUARED WITH DIMENSIONS POWER OF TWO WITHIN A RANGE BETWEEN 64 AND 2048.

Zernike	Fermi GPU						Kepler GPU						Improvement factor	
moments	64	128	256	512	1024	2048	64	128	256	512	1024	2048	Minimum	Maximum
$A_{4,*}$	0,12	0,17	0,37	1,11	4,08	15,75	0,18	0,19	0,33	0,61	1,91	7,17	0,67x	2,20x
$A_{8,*}$	0,20	0,32	0,74	2,36	8,83	34,71	0,28	0,31	0,46	1,16	3,86	14,74	0,71x	2,35x
$A_{12,*}$	0,29	0,50	1,21	4,08	15,32	60,41	0,41	0,44	0,72	1,88	6,42	24,67	0,71x	2,45x
$A_{16,*}$	0,38	0,72	1,82	6,14	23,49	92,05	0,53	0,58	1,00	2,72	9,55	36,93	0,72x	2,49x
$A_{20,*}$	0,51	0,97	2,50	8,68	33,37	130,93	0,67	0,74	1,33	3,75	13,28	51,56	0,76x	2,54x
$A_{24,*}$	0,61	1,27	3,31	11,76	45,39	176,51	0,80	0,90	1,70	4,92	17,64	68,59	0,76x	2,57x
$A_{28,*}$	0,74	1,59	4,23	15,20	58,20	229,20	0,97	1,09	2,11	6,23	22,52	87,87	0,76x	2,61x
$A_{32,*}$	0,87	2,00	5,27	18,89	73,30	288,76	1,14	1,29	2,58	7,70	28,04	109,53	0,76x	2,64x
$A_{34,*}$	0,95	2,16	5,89	20,96	81,29	319,76	1,22	1,39	2,81	8,48	31,01	121,23	0,78x	2,64x

On a vertical analysis, the order increment in Zernike moments to be computed requires an additional execution of the algorithm for every couple of orders. This additional workload increases the computational time with a slight benefit for the GPU in the global process.

B. Workload set up

Raising the number of cores within the SMX multiprocessor in a 6x factor with respect to SM in Fermi poses a question about the optimal set up for the size of the block in the CUDA grid. On Fermi GPUs endowed with SMs of 32 cores, a value between 128 and 256 threads was optimal to achieve 100% occupancy as long as there were no restrictions about the use of registers and shared memory. On Kepler GPUs with SMXs having 192 cores, this requires a more demanding analysis.

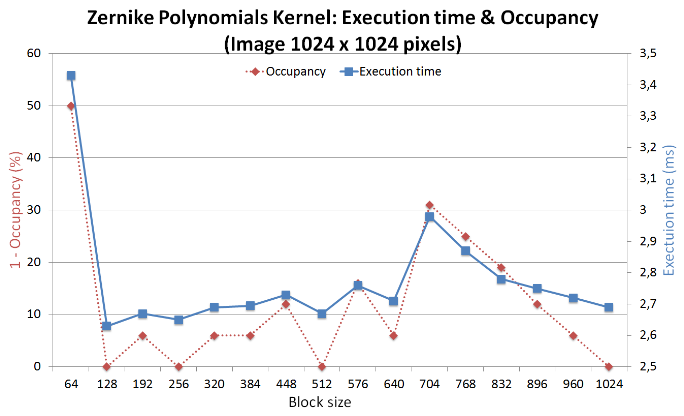


Fig. 2. Execution time on Kepler to evaluate performance (on the right scale and using a blue solid line) as a function of the CUDA block size. Those values are compared with the theoretical occupancy for the architecture on the left scale using a red dashed line.

Figure 2 unveils the execution times for the kernel computing the Zernike polynomials, that being the critical kernel as long as resources is concerned. Times in the right scale (solid line) are compared with the theoretical ones for blocks of different sizes, leading to the occupancy depicted on the left scale (dotted line). Even though peak performance is usually attained for block sizes of a power of two from 128 threads on, our results are slightly better for an exact block size of 128 threads, the lower among the potential candidates. However, the bigger divergence between theory and practice corresponds to those cases when the number of threads per block is not enough to exploit resources, and also when an additional block exceeds the limit of threads to be simultaneously executed on a single multiprocessor, leading to a block sacrificed from concurrent execution.

1) If the block contains less than 128 threads, the limitation for not reaching the maximum number of threads per multiprocessor is imposed by the maximum number of blocks scheduled. On Kepler GPUs, we can execute up to 16 active blocks on each SMX multiprocessor, and the formula for occupancy takes the following expression:

$$Occupancy = \frac{Threads * 16}{2048} \tag{8}$$

2) When the number of threads allocated on a multiprocessor reaches its maximum value, an additional block may exceed this maximum and, therefore, the number of allowed blocks does not exploit all hardware resources. The worst case scenario (69% occupancy) corresponds to a set up of 704 threads per block where only two active blocks can be scheduled (three blocks lead to 2112 threads, exceeding the maximum of 2048 on each multiprocessor). In general, the following expression can be applied to calculate the percentage of occupancy:

$$Occupancy = \frac{BlockSize * NumberOfActiveBlocks}{2048} \tag{9}$$

C. Dynamic parallelism

The use of dynamic parallelism that we described in section VI-B is going to be evaluated here to discuss when its application is profitable. Table IV(a) shows that its simplest use, tagged "Launch kernels from the GPU", and the strategy "Calculate a repetition from each thread" are not profitable. Those results were taken for moments with specific orders and repetitions to give an idea about the execution times for

TABLE IV
EXECUTION TIMES AND ACCELERATION FACTORS ATTAINED FOR DIFFERENT STRATEGIES MAKING USE OF DYNAMIC PARALLELISM ON SQUARE IMAGES OF DIFFERENT SIZES. THE UPPER TABLE SHOWS SELECTED ZERNIKE MOMENTS, THE LOWER ONE SWEEPS OVER ALL REPETITIONS FOR A GIVEN ORDER.

(a) Execution times in milliseconds without/with dynamic parallelism.

Zernike	Dyn. par. disabled			Dyn. par. enabled			Performance gain	
moment	64	256	1024	64	256	1024	Min.	Max.
$A_{0,0}$	0,08	0,10	0,56	0,16	0,20	0,88	0,48x	0,64x
$A_{6,2}$	0,08	0,13	0,93	0,16	0,22	1,26	0,53x	0,74x
$A_{12,0}$	0,09	0,16	1,43	0,16	0,27	1,79	0,58x	0,80x
$A_{25,13}$	0,09	0,17	1,52	0,16	0,27	1,88	0,59x	0,81x
$A_{34,0}$	0,12	0,28	3,07	0,18	0,38	3,51	0,65x	0,88x
$A_{34,18}$	0,10	0,19	1,82	0,16	0,29	2,20	0,60x	0,83x
$A_{34,34}$	0,08	0,10	0,63	0,16	0,20	0,95	0,49x	0,66x

(b) Execution times in milliseconds using dynamic parallelism.

All moments	Image size					
for a given order	64	128	256	512	1024	2048
$A_{4,*}$	0,73	0,70	0,78	0,70	0,71	0,72
$A_{8,*}$	0,82	0,76	0,71	0,75	0,76	0,76
$A_{12,*}$	0,89	0,82	0,76	0,79	0,79	0,79
$A_{16,*}$	0,91	0,84	0,78	0,80	0,81	0,81
$A_{20,*}$	0,93	0,86	0,81	0,82	0,82	0,82
$A_{24,*}$	0,93	0,87	0,82	0,84	0,84	0,84
$A_{28,*}$	0,95	0,89	0,84	0,85	0,85	0,85
$A_{32,*}$	0,96	0,91	0,85	0,86	0,86	0,86
$A_{34,*}$	0,96	0,90	0,85	0,86	0,86	0,86

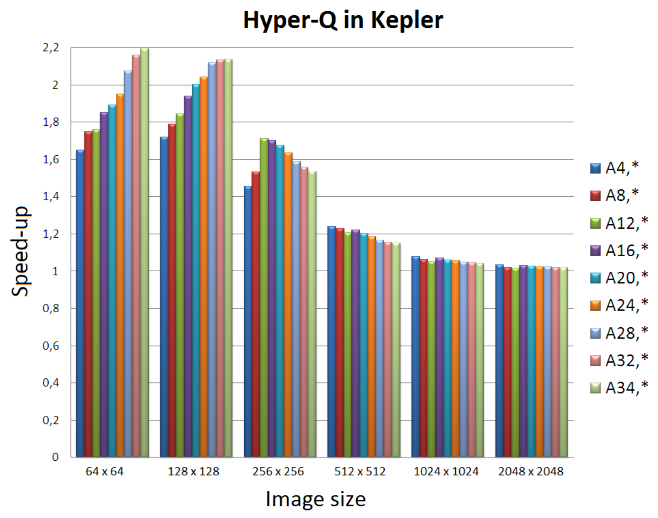
individual moments in addition to compare the results with dynamic parallelism.

In the first case, execution times increase a factor between 1.15x ad 2.15x (see Table IV(a)). This variance depends on the overhead for calling a new kernel from the GPU. The penalty is lower on larger image sizes and moments more demanding computationally (those with a higher difference between the order and its repetition).

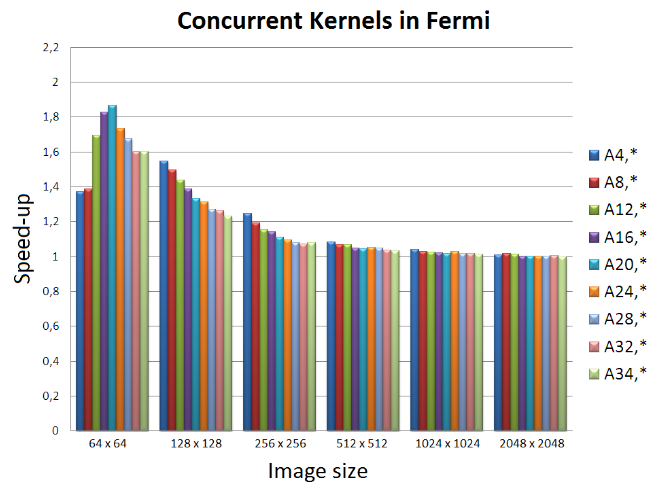
For the second strategy, performance worsens up to a 1.4x factor following the previous strategy (see Table IV(b)). In this case, performance is better on small images due to processing a set of moments which make a better use of hardware resources. This would not be possible for isolated moments. In summary, performance on small images is sensitive to two main aspects: launching kernels from the GPU itself, and the parallel execution for a collection of moments characterized by a chain $A_{n,*}$ (the whole set of repetitions required by an order n).

The third strategy, called "Parallelize the loop of the Zernike polynomials" is quite ambitious for its dynamic nature. However, experimental tests show immediately that performance here is very poor. The implementation requires each thread to launch a new kernel, and time skyrockets to reach 1000x factors quickly. We have measured the time consumed for each GPU kernel call, obtaining values between 5 and 16 μsecs., whereas the kernel call executed from the CPU side consumes around de 3 μsecs. It does not seem logical to assume that the kernel launch introduces more overhead when it is generated from circuitry much closer to the GPU, and we believe that this slowdown will be solved on more mature versions of the drivers and/or subsequent enhancements on SMX multiprocessors.

Nevertheless, dynamic parallelism is oriented to applications following a "divide and conquer" strategy, and must be used with similar guidelines already popular in GPUs: Making few calls to kernels dealing with a huge amount of data. Another feature shortening the scope of application for dynamic parallelism is the constraint that each thread cannot access the shared memory space of its father kernel. The information to be shared among father and sons are delegated to global memory space, incurring a performance penalty which sometimes is decisive.



(a) Improvements achieved with Hyper-Q on Kepler.



(b) Improvements achieved with concurrent kernels in Fermi.

Fig. 3. Performance gains with the use of CUDA streams for different image sizes when we raise the set of Zernike moments to compute.

D. Hyper-Q

In order to take advantage of Hyper-Q in our algorithm, we define a stream for each repetition when we aim to compute all repetitions for a given order. Figure 3(a) compares the acceleration factor when Hyper-Q is enabled under these assumptions, varying the order for the Zernike moments and the size of the input images.

The maximum gain obtained is 2.2x, whereas our worst performance result is a draw with the basic implementation (no speed-up). This parity situation arises for a workload capable of filling all hardware resources. If the image size is such that keeps all GPU resources busy, there are no leftovers to be exploited by additional streams via Hyper-Q, and those streams will end up executed sequentially on working queues. On the other hand, on smaller image sizes, gains increase lineally with the number of repetitions to compute for each moment. The maximum performance is attained when processing the smallest image and the Zernike moment of highest order. This scenario is ideal for exploiting Hyper-Q, as the little workload required by each image can be compensated by eventually allowing additional images to be processed in parallel.

Since using Hyper-Q does not require any change from the developer's side and the queues manager to process the streams is transparent, the same implementation executed on Fermi hardware provides us some insights about the improvements that were brought to the GPU when concurrent kernels were enabled. Figure 3(b) shows, following what we did in Kepler, the gain factor obtained when we count on that feature in Fermi. Performance results are similar, but this time the smaller image size maximizes performance on lower moment orders because it is already enough to reach full occupancy of hardware resources. In particular, the maximum factor is 1.86x on 11 streams, a situations which arises for the computation of the twentieth moment (A_{20}).

Hyper-Q has contributed with a maximum speed-up of 1.74x in addition to what we had already attained using concurrent kernels in Fermi.

The previous analysis combines those improvements introduced by Hyper-Q with those due to the full occupancy of hardware resources on GPU multiprocessors. Now we pretend to isolate the acceleration provided by Hyper-Q by conducting an experiment on a 16 x 16 image size (which matches our CUDA block size). This way, a conventional execution of all repetitions for a given order is performed sequentially and without taking advantage of all resources (a single block is sent per iteration). When using Hyper-Q or concurrent kernels, resources usage increases due to the parallel execution of repetitions.

Figure 4 shows the results comparing Fermi and Kepler under those assumptions. Fermi is ahead in performance when the parallelization does not use streams (the image size is too small). This result was already explained in section VII-A on images of 64x64 pixels. When Hyper-Q is enabled in Kepler or concurrent kernels are used in Fermi, the gain factor is higher in Kepler (1.7x), even though its execution time remains slower than Fermi. This make us believe that Hyper-Q is a great partner to exploit the increment in number of processing cores, but does not contribute much to the pillars deploying parallelism in CUDA, namely, blocks of threads within multiprocessors and grids of concurrent blocks among multiprocessors. Our hypothesis will be validated after the recursive approach of the Zernike moments be analyzed in the following section.

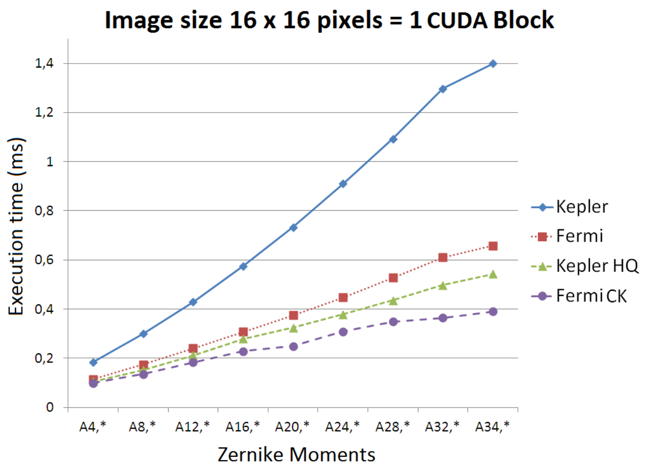


Fig. 4. Comparative analysis for the improvements attained by the GPU when the input image has 16 x 16 pixels, which correspond to a CUDA block of threads. This pretends to isolate the benefit obtained by Hyper-Q and concurrent kernels in our execution.

E. Recursive methods

Until now, we have exploited dynamic parallelism and Hyper-Q to compute all repetitions for a given order of Zernike moments. This is the easiest way to increase the amount of data to compute and skip data dependencies, but section VI-A described recursive algorithms too. If we explore this path, we will definitely have a faster execution on the CPU, but less opportunities to apply parallelism on the GPU.

Table V shows the execution times obtained when we apply the *q-recursive* method on Fermi and Kepler GPUs, and compares those with respect to the direct methods formerly executed, where chances for deploying parallelism are higher. Results favor the recursive implementation in almost every case, with the exception of the moment of lowest order and the smallest image. Accelerations grow as we increase the moment order and the image size, and so Fermi does the same. We find two reasons to justify this amazing result: First, the recursive implementation reduces the complexity of the operations, and therefore, the GPU workload, which penalizes the Kepler platform. Second, the reference time is higher in Fermi, and this departure point grants higher potential for improvements in this platform.

F. Recursion versus Hyper-Q in direct methods

Previous sections tackled Zernike moments from two different perspectives: Using direct methods, which allow to

TABLE V
EXECUTION TIMES (IN MILLISECONDS) FOR SQUARE IMAGES OF DIFFERENT SIZES VIA THE *q-recursive* METHOD ON FERMI AND KEPLER GPUS. GAINS ON THE LAST FOUR COLUMNS ARE CALCULATED TAKING AS REFERENCE THE TIME ELAPSED BY THE DIRECT METHOD.

Zernike	Fermi GPU						Kepler GPU						Gain on Fermi		Gain on Kepler	
moments	64	128	256	512	1 K	2 K	64	128	256	512	1 K	2 K	Min	Max	Min	Max
$A_{4,*}$	0,08	0,12	0,29	0,90	3,31	13,07	0,11	0,12	0,24	0,59	2,01	7,76	1.21	1.50	0.92	1.62
$A_{8,*}$	0,11	0,17	0,45	1,50	5,78	22,53	0,14	0,16	0,32	0,91	3,27	12,73	1.53	1.92	1.16	2.05
$A_{12,*}$	0,12	0,23	0,61	2,12	8,12	32,12	0,17	0,20	0,42	1,25	4,54	17,73	1.88	2.35	1.39	2.34
$A_{16,*}$	0,14	0,28	0,79	2,74	10,58	41,66	0,20	0,24	0,52	1,58	5,82	22,79	2.21	2.67	1.62	2.58
$A_{20,*}$	0,17	0,34	0,95	3,36	13,13	51,35	0,23	0,29	0,62	1,91	7,11	27,87	2.54	3.07	1.85	2.88
$A_{24,*}$	0,19	0,39	1,12	3,98	15,59	60,88	0,26	0,32	0,72	2,24	8,38	32,93	2.90	3.26	2.08	3.06
$A_{28,*}$	0,21	0,45	1,28	4,60	17,97	70,56	0,28	0,36	0,82	2,58	9,66	38,03	3.24	3.56	2.31	3.42
$A_{32,*}$	0,23	0,50	1,45	5,31	20,41	80,19	0,32	0,41	0,93	2,92	10,95	43,15	3.56	4.02	2.54	3.52
$A_{34,*}$	0,24	0,52	1,54	5,55	21,64	85,00	0,33	0,44	0,98	3,09	11,59	45,70	3.76	4.12	2.65	3.66

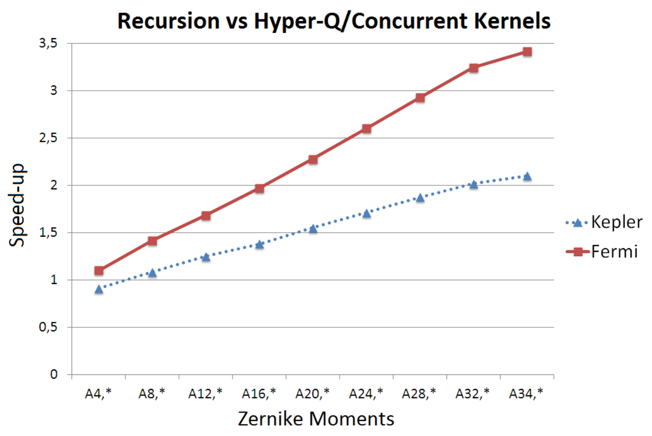


Fig. 5. Experimental comparison between the recursive method and the direct method using Hyper-Q. Those values greater than 1 mean higher performance on the recursive side.

apply higher degree of parallelism, and recursive methods, which amortize the computational cost of previous orders and repetitions. A final comparison between those two strategies will determine the winner technique in the GPU arena.

Figure 5 shows the comparison between the two methods, a battle where brute force and smart computation meet. Recursive methods win on Fermi and Kepler, showing a scalable and linear behavior with the moment order for the whole range considered (up to an order of 34).

Moreover, larger image sizes benefit to Kepler GPUs. This is because SMX multiprocessors use a larger number of GPU cores when a conventional workload distribution is applied via the CUDA grid. Once we have reached the necessary data volume to feed all computational cores, the overhead introduced by a more sophisticated Hyper-Q when managing kernels and streams is clearly amortized.

The main conclusion we draw from this analysis is that the parallelism deployment inherent to algorithms must be focused on exploiting intra-block concurrency (among the cores of each multiprocessor) and the number of active blocks (among the multiprocessors of a GPU), leaving the third level of concurrent kernels and Hyper-Q as a third party when the workload does not allow to saturate the high number of computational cores and resources. This situation often arises within the context of irregular applications where data volume is spread on a huge set of small tasks where each of them deals with little data structures, say lower than a thousand elements. It will also be a likely assumption in future hardware generations endowed with larger number of cores, which indicates that we are just at the beginning of a long way road to exploit the potential hidden by Hyper-Q as a parallel schedule resource.

G. Software needs versus hardware availability

The hardware evolution from the G80 (the first CUDA-enabled GPU) to Fermi and later Kepler has been driven by an astonishing increment in the number of functional units, computational cores dedicated to integer arithmetic (ALUs), floating-point computations (FPUs), special function units (SFUs) and load/store units. A tough question to be answered by hardware designers is how many cores they dedicate to each kind of computation, and how well those percentages fit into the software needs of every particular CUDA code. Algorithms using units in the right proportion will exploit better the GPU versus those persistently using the same type of functional unit at the expense of leaving others idle.

We have wondered this for our case study, and Table VI gives the appropriate answer for our second and third CUDA kernels when executed on Fermi and Kepler. We have chosen these two kernels because the first one is too simple, and the fourth and fifth ones implement reduction operators which have been extensively studied in the literature.

In the second kernel, Kepler dedicates less percentage of units to double precision and more to special function units, which is good for the code requirements. On the contrary, it generates a higher bottleneck on load/store units and exceeds more the availability of integer arithmetic and single precision FPUs. Overall, Fermi fits better into what this kernel computationally demands. But fortunately, memory bandwidth comes to a rescue here, as it has been increased from 36.6 GB/s in Fermi to 468.1 GB/s in Kepler, which means higher

TABLE VI
HARDWARE AVAILABILITY OF GPU RESOURCES VERSUS THE USE THAT OUR CUDA KERNELS MAKE OF THEM. WE EVALUATE OUR SECOND AND THIRD KERNELS ON FERMI AND KEPLER GPUS.

GPU resources	ALUs	Single Precision FPUs	Double Precision FPUs	Load/Store Units	Special Function Units
Offered by Fermi SM multiprocessors	32%	32%	16%	16%	4%
Offered by Kepler SMX multiprocessors	37.5%	37.5%	12.5%	6.25%	6.25%
Required by our second CUDA kernel	23.7%	30.9%	0.0%	21.0%	24.4%
Who fits better into our second kernel needs	Fermi	Fermi	Kepler	Fermi	Kepler
Required by our third CUDA kernel	22.2%	22.2%	0.0%	55.6%	0.0%
Who fits better into our third kernel needs	Fermi	Fermi	Kepler	Fermi	Fermi

throughput for every kind of instruction, particularly when the programmer makes an extensive use of shared memory.

Another path to explore at this point is to avoid the use of intrinsic CUDA instructions to delegate all math operators onto conventional floating-point units, leaving aside SFUs. When this variant is accomplished in Kepler, performance decreases as the workload raises, converging to a 3x factor. And the slowdown is even higher in Fermi for having less percentage of single precision FPUs. Such a remarkable difference in performance justifies the presence of SFUs in SMX multiprocessors, having increased from 4 in Fermi to 32 in Kepler. Even though these special math functions are not widely used, the computation of a counterpart version on typical FPUs use classic algorithms decomposed into polynomial approaches that are way inefficient.

Finally, the third kernel represents a simpler case where both platforms are bandwidth limited, reaching 188.7 GB/s and 120.7 GB/s for Kepler and Fermi, respectively. Kepler increases bandwidth more than 50% here, which is profitable for issuing warps more aggressively on the front-end (Kepler can issue up to eight warp-instructions coming from four different warps, whereas Fermi can only issue a pair). Looking at the back-end, Fermi contains a higher percentage of load/store units, but Kepler increases the number up to the warp size. That is the optimal number for the active warps consuming the video memory bandwidth or when playing with the 32 banks of shared memory. Wasted resources are double precision FPUs and SFUs this time, with a percentage around 20% of total hardware resources deployed by Fermi and Kepler. Integer and single precision floating-point arithmetic needs match (22.2% each), being Fermi closer than Kepler (32% for those type of units versus 37.5% provided by Kepler).

Overall, thanks to the higher memory bandwidth, the front-end and back-end seem quite compensated in Kepler for the computational requirements of our kernels, whereas Fermi offers a better balance of computational units inside the back-end and shows shortages on the front-end for issuing instructions when the kernel is embarrassingly parallel.

VIII. CONCLUSIONS

This work analyzes the GPU capabilities for accelerate the computation of Zernike moments, focusing on the Kepler architecture and its SMX multiprocessor. The architectural changes from the previous Fermi generation are remarkable and have allowed us to improve performance up to 265% using the same implementation, with higher gains relying on dynamic parallelism and Hyper-Q. Dynamic parallelism was not profitable due to an overhead detected on the mechanisms for launching kernels internally from the GPU. We expect this shortcoming to be fixed as Kepler hardware evolves. On the positive side, Hyper-Q increases performance up to 220% with respect to the baseline implementation of direct methods in Kepler, producing better results when the workload coming from a single kernel does not saturate the number of cores available on the GPU. This gain can be improved in parallel codes using POSIX threads or even MPI.

The recursive method is improved on the Kepler architecture even though we sacrifice most of the parallelism to deploy on the GPU and Hyper-Q benefits. As compared to direct methods, results worsen only up to an order of six, improving from that Zernike moment on to reach percentages of 350%.

Ultimately, the best method for computing Zernike moments on the GPU depends on our application idiosyncrasy. A broad set of problems like image classifiers and texture characterization require only a small number of selected moments (those showing better discrimination properties after a preliminary analysis). Other examples like image compression and reconstruction rely on the whole set of repetitions and orders of high degree where the recursive formulation can be applied on a second dimension in our parameters space. This work studies a hybrid situation which considers the computation of all repetitions for a given order, finding the basic criteria to select the more advantageous implementation on GPUs: Either using direct methods thanks to the parallelization mechanisms in CUDA, or via the recursive formulation despite of its good behavior on CPUs. And our results are more competitive when the input image grows in size and the underlying hardware increases the number of functional units available, two factors which are expected to follow a scalable evolution for the software applications and hardware platforms yet to come.

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education of Spain under projects TIN2009-14475-C04 and by the Junta de Andalucía under Project of Excellence P12-TIC-1741. We also thank NVIDIA for hardware donation under CUDA Teaching Center 2011-14, CUDA Research Center 2012-14 and CUDA Fellow 2012-14 Awards.

References

[1] Y. Bin and P. Jia-Xiong, "Invariance analysis of improved Zernike moments," *Journal of Optics A: Pure and Applied Optics*, vol. 4, no. 6, p. 606, 2002.

[2] A. Khotanzad and Y. H. Hong, "Invariant image recognition by Zernike moments," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 12, no. 5, pp. 489–497, 1990.

[3] Y. S. Abu-Mostafa and D. Psaltis, "Recognitive aspects of moment invariants," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 698–706, 1984.

[4] M. R. Teague, "Image analysis via the general theory of moments," *J. Opt. Soc. Am*, vol. 70, no. 8, pp. 920–930, 1980.

[5] Ø. Due Trier, A. K. Jain, and T. Taxt, "Feature extraction methods for character recognition - a survey," *Pattern recognition*, vol. 29, no. 4, pp. 641–662, 1996.

[6] C.-H. Teh and R. T. Chin, "On image analysis by the methods of moments," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 10, no. 4, pp. 496–513, 1988.

[7] R. Mukundan and K. Ramakrishnan, "Fast computation of Legendre and Zernike moments," *Pattern recognition*, vol. 28, no. 9, pp. 1433–1442, 1995.

[8] M. Al-Rawi, "Fast Zernike moments," *Journal of Real-Time Image Processing*, vol. 3, no. 1-2, pp. 89–96, 2008.

[9] M. J. Martín-Requena and M. Ujaldón, "Leveraging graphics hardware for an automatic classification of bone tissue," in *Computational Vision and Medical Image Processing*. Springer, 2011, pp. 209–228.

[10] M. J. Martin-Requena and M. Ujaldon, "High performance computation of moments for an accurate classification of bone tissue images," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. IEEE, 2011, pp. 593–598.

[11] C.-W. Chong, P. Raveendran, and R. Mukundan, "A comparative analysis of algorithms for fast computation of Zernike moments," *Pattern Recognition*, vol. 36, no. 3, pp. 731–742, 2003.

[12] S.-K. Hwang and W.-Y. Kim, "A novel approach to the fast computation of Zernike moments," *Pattern Recognition*, vol. 39, no. 11, pp. 2065–2076, 2006.

[13] NVIDIA, "Parallel programming and computing platform," http://www.nvidia.es/object/cuda_home_new_es.html, Jun. 2013.

[14] NVIDIA, "The Kepler architecture," http://www.nvidia.com/object/nvidia-kepler.html, Jun. 2013.