Optimized CUDA-based PDE Solver for Reaction Diffusion Systems on Arbitrary Surfaces

Samira Michèle Descombes[1], Daljit Singh Dhillon[1], and Matthias Zwicker[1]

Institute of Computer Science, University of Bern, Bern, Switzerland.
samira.descombes@students.unibe.ch, djdhillon@gmail.com,
zwicker@inf.unibe.ch,
WWW home page: http://www.cgg.unibe.ch/

Abstract. *Partial differential equation* (PDE) solvers are commonly employed to study and characterize the parameter space for *reaction-diffusion* (RD) systems while investigating biological pattern formation. Increasingly, biologists wish to perform such studies with arbitrary surfaces representing 'real' 3D geometries for better insights. In this paper, we present a highly optimized CUDA-based solver for RD equations on triangulated meshes in 3D. We demonstrate our solver using a *chemotactic* model that can be used to study snakeskin pigmentation, for example. We employ a *finite element* based approach to perform *explicit Euler time integrations*. We compare our approach to a naive GPU implementation and provide an in-depth performance analysis, demonstrating the significant speedup afforded by our optimizations. The optimization strategies that we exploit could be generalized to other mesh based processing applications with PDE simulations.

Keywords: CUDA · GPU programming · Reaction-Diffusion Systems · Nonlinear PDEs · FEM · Explicit time-stepping

1 Introduction

Partial differential equations defined on triangulated meshes in 3D play an important role in many applications. In 3D geometry processing, for example, 3D shapes are smoothed and denoised via curvature flow, or heat diffusion on surfaces can be used to define geometric features. Physical effects such as deformation or cracking can also be modeled using PDEs on surfaces. In this paper we are considering reaction-diffusion equations, which are widely believed to play a crucial role in biological pattern formation. For these applications, efficient PDE solvers on triangulated surfaces are crucial components. To address this need, the goal of this paper is to develop an optimized GPU implementation for one type of surface PDEs, in particular reaction-diffusion equations, and study the improvements that can be achieved over a naive approach by careful profiling.

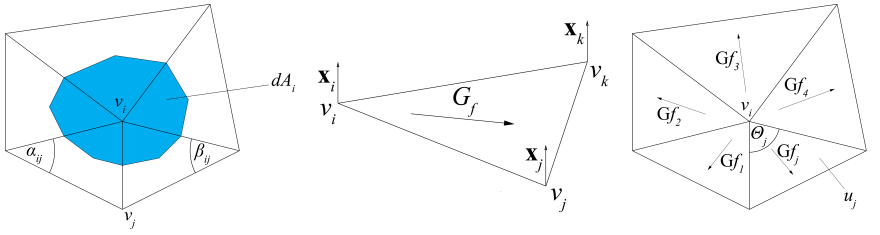
Reaction-diffusion (RD) models have first been hypothesized by Turing as a mechanism that is involved in biological pattern formation. For biologists

it is important to study these equations on 3D geometries to obtain better insights about their behavior in realistic scenarios. In this paper we employ a finite element based approach and perform explicit Euler time integration to solve these equations on triangulated meshes in 3D. The discretization and time integration lead to simple discrete operators that we need to evaluate locally on small neighborhoods of mesh vertices. This offers the possibility of a straightforward parallel implementation on GPUs, where the discrete operators are computed simultaneously on many vertices. We develop a more sophisticated, optimized implementation, however, that demonstrates how techniques such as kernel fusion, partitioning the input data, and effective use of shared memory and device arrays provides significant further performance gains. A contribution of this paper is to show how these techniques could also be leveraged to implement other mesh based PDE solvers efficiently on GPUs. We discuss how the different degrees of freedom that can be exploited to optimize a mesh based PDE solver could be leveraged more generally, for example by exposing them in a domain-specific language that targets this problem. Our discussion is inspired by the recent development and popularity of Halide, a domain-specific language for image processing.

2 Related Work

Partial differential equations based on reaction-diffusion, or Turing [14], models have been widely postulated to be relevant in biological pattern formation, and experimental evidence supporting this hypothesis has been found in various instances. Pioneering work by Kondo and Asai [8] showed for the first time how a simulation of a reaction-diffusion model correctly predicts skin patterns on the marine angelfish, *Pomacanthus*. For an excellent recent survey we refer to the review by Kondo and Miura [9].

There is an abundance of libraries and frameworks for mesh based PDE solvers that target a variety of systems, ranging from GPUs to large-scale clusters. Libraries like PETSc [3] or Sandia's Sierra [2] build on the SPMD model to enable programmers to write code that largely resembles single-threaded programs, while execution may be distributed over a variety of processors. Similarly, Liszt's [5] concept of "for-comprehension" allows the system to choose a parallel implementation, while hiding this from the program code. Other frameworks [4, 6] rely on the concept of kernels to express computations that need to be executed on a set of data elements. While some of these systems include back-ends for GPU code generation, they do not easily allow the user to optimize the performance of GPU code. In contrast, Halide [12] is a domain-specific language for image processing that also abstracts away the parallel execution from the specification of an algorithm. A key idea is that it allows the user to provide a so-called schedule in addition to the algorithm, which is a high level description of a desired parallel execution configuration. By adjusting the schedule a user can obtain highly optimized implementations for various back-ends including multicore CPUs and GPUs. Halide is restricted to operate on the rectangular



(a) Laplace-Beltrami (b) Face gradient (c) Vertex gradient

Fig. 1: Illustrating discrete geometry operations

topology of images, however, and it does not consider general meshes. We believe our optimizations could be exposed in a similar high-level language, but targeted at processing data on meshes.

3 Computational Model

We first present a chemotactic RD model that has been proposed for snakeskin pattern formation [11]. We then explain the use of discrete geometric constructs to simulate this PDE based system on an arbitrary mesh. Finally, we give the details of the simulation steps.

Reaction-Diffusion with Chemotaxis. The chemotactic model proposed by Murray et al. [11] is expressed mathematically in a dimensionless form as,

$$\begin{aligned}\frac{\partial f_n}{\partial t} &= D\nabla^2 f_n - \alpha f_n \nabla^2 f_c - \alpha \nabla f_n \cdot \nabla f_c + s r f_n (N - f_n)\,,\\ \frac{\partial f_c}{\partial t} &= \nabla^2 f_c + s\left(\frac{f_n}{1+f_n} - f_c\right).\end{aligned} \tag{1}$$

Here, f_n is a non-negative cell density function, f_c is a non-negative chemoattractant density function, ∇^2 is the Laplacian operator, ∇ is a gradient operator, D a positive cell diffusion rate, α a positive chemotactic rate, r the cell growth rate, s a positive scale factor, and N the maximum cell density capacity. The PDEs operate on a restricted region within the snakeskin surface, subject to zero Neumann boundary conditions. For further discussion of the cell-chemotaxis model see the work by Murray [11] and Winters et al. [1].

FEMs and Discrete Geometry. We use discrete geometric operators to simulate Equation 1 on an arbitrary surface. The Laplacian ∇^2 is replaced by a Laplacian-Beltrami operator that is evaluated for a surface function f around a mesh vertex v_i as,

$$\nabla^2 f(v_i) = \frac{1}{2A_i} \sum_{v_j \in N_1(v_i)} (\cot \alpha_{ij} + \cot \beta_{ij})(f(v_j) - f(v_i))\,. \tag{2}$$

Here, A_i is the area of the averaging region dA_i around vertex v_i and $(\cot \alpha_{ij} + \cot \beta_{ij})$ is called cotangent weighting. Figure 1a illustrates the calculation. Reuters et al. [13] state that using cotangent weights with the normalizing weights as A_i's is equivalent to a lumped-linear FEM formulation for the Laplace eigenproblem. We thus use these weights as expressed in Equation 2. Next we compute the function gradients in two steps: (a) computing face-gradients from the function values at vertices, and (b) computing vertex-gradients at the vertices by appropriate weighted averaging over face-gradients. For a face u_i consisting of vertices v_i, v_j and v_k the face-gradient is given by,

$$\nabla f(u_i) = (f(v_j) - f(v_i))\frac{(\mathbf{x}_i - \mathbf{x}_k)^{\perp}}{2A_T} + (f(v_k) - f(v_i))\frac{(\mathbf{x}_j - \mathbf{x}_i)^{\perp}}{2A_T}, \qquad (3)$$

where $\mathbf{x}$ is the 3D coordinate of vertex v, A_T is the area of the face and $\perp$ is a counterclockwise rotation of a vector by 90°, at its tail, in the triangle plane. The definition is illustrated in Figure 1b. Next, the gradient for each vertex v_i is computed as,

$$\nabla f(v_i) = \sum_{u_j \in nbr(v_i)} \nabla f(u_j) w_j \,. \qquad (4)$$

Here $nbr(v_i)$ is a set of faces u_j incident on vertex v_i and w_j are the normalized weights for neighborhood-averaging, see Figure 1c. With an appropriate choice of the weights in Equations 3 and 4, these computations approximate first order differentials well [10]. We simply use the incident angles θ_j to generate normalized weights for Equation 4 since this is appropriate in a discrete geometric sense.

Euler Integration. For our simulation we use explicit time-stepping. For a system state $Y(t)$ defined by (f_n, f_c), we calculate the state at later time $Y(t + \delta t) = Y(t) + \delta t(\partial Y/\partial t)$, where δt is a small time-step. Each simulation begins with: (a) *An initialization* with parameters $D = 0.25$, $r = 1.522$, $\alpha = 12.02$, $s = 1$ and $N = 1$ unless otherwise specified. Also $f_n = N$ and $f_c = N/(1 + N)$ and we add minor random perturbations to both. This is followed by: (b) *A transition loop* where all gradients are computed using discrete operations, as discussed earlier, to evaluate the time-derivatives in Equation 1 and to update the system state $Y(t)$ through explicit time-stepping. Finally: (c) we terminate the iterations in the transition loop upon reaching convergence or a predetermined number of time-steps. We use time-steps δt of a fixed size that is small enough and $\mathcal{O}(dx^2)$, where dx is the average edge length for the given mesh.

4 GPU Optimizations

In this section we describe in detail our use of various GPU optimization techniques and state their overall impact briefly. Later, in Section 5 we provide an in-depth performance analysis for each of the techniques to understand their influence on the speedup.

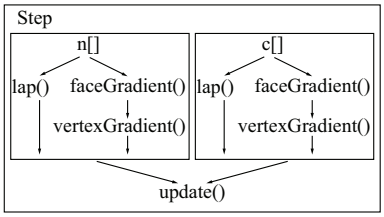
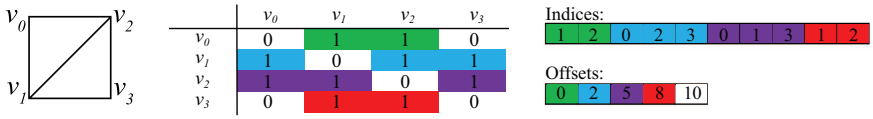
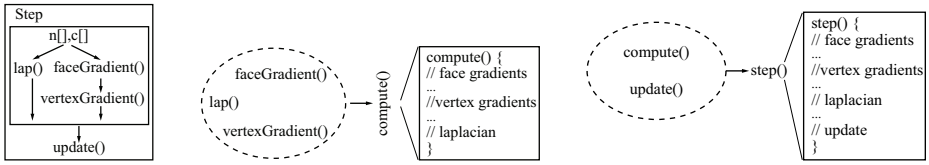


Fig. 2: The *baseline* implementation



(a) Graph (b) Adjacency matrix (c) Compressed sparse row

Fig. 3: Storing neighborhood references



(a) *Fusion 1* (b) *Fusion 2* (c) *Fusion 3*

Fig. 4: Implementation outline for *Fusion 1*, *Fusion 2* and *Fusion 3*

Naive baseline implementation. We begin with a *baseline* GPU implementation for our iterative transition loop with four CUDA kernels as shown in Figure 2. Kernel *lap()* computes surface Laplacians for functions f_n and f_c as expressed in Equation 2. We then compute gradients with kernels *faceGradients()* and *vertexGradients()* implementing Equations 3 and 4 respectively. Finally, we compute the time-derivatives and update the system state $Y(t) \equiv (f_n, f_c)$ in kernel *update()* with an explicit time-step. Our *baseline* implementation uses: (a) one thread per vertex, (b) a compressed sparse row (CSR) representation for vertex connectivity (see Figure 3c), (c) only the memory registers and the global memory. This implementation incurs huge memory overloads due to uncoalesced, repetitive memory accesses for the context data and intermediate results stored in the global memory. Also, it experiences frequent thread stalls due to divergent executions. The naive *baseline* implementation still provides a speedup of about 125× with single-precision operations compared to a single-threaded CPU implementation.

Our optimized implementation. Mainly there are two areas that deserve consideration for optimizations, namely: (a) computational flow, and (b) memory accesses. We perform three kernel fusions as depicted in Figure 4 to optimize the computational flow and use *shared memory* and *device arrays* to optimize memory accesses. With these optimizations, we improve performance by a multiplicative factor of about 4 over the *baseline* version. This gives overall improvements on the order of 500× for the single-precision version and 350× with double-precision operations.

Kernel fusions. The *baseline* computation flow has two major areas of improvement: (a) shared input data, mainly the function values, are loaded repeatedly for each kernel and (b) intermediate data resides in global memory. We first improve by fusing together the computation kernels for f_n and f_c. This reduces kernel

invocations from 7 to 4 (see Figure 4a). Then, we merge the *computing kernels* together (see Figure 4b) and finally, we merge the remaining two kernels into one, as shown in Figure 4c. *Fusion 1* results in 1.5× performance gains but *Fusion 2 & 3* does not give immediate gains. However, *Fusion 2* leads to important data reorganization as explained below, which ultimately reduces data access time considerably with memory optimizations.

Mesh partitioning. Fusion 2 is problematic since kernel *vertexGradient()* depends on the output of kernel *faceGradient()* and this warrants a synchronization. Within a single kernel however, only block-wide synchronization is possible which is insufficient. Consider the computation flow for the blue block in Figure 5 where only block-wise synchronization leads to erroneous results in vertex-gradients for shared vertices in adjoining orange block. We thus partition the mesh such that each block is associated to a locally fully-connected *patch*. The process is illustrated by Figure 6. Partitioning results in a few *halo* faces, shown in Figure 6c that are added to multiple blocks which add few redundant computations to our computational flow. These overheads are negligible in comparison with later gains due to memory optimizations. We use the METIS library [7] to perform offline multilevel mesh partitioning to: (1) produce partitions of near equal size and (2) minimize the number of halo faces.

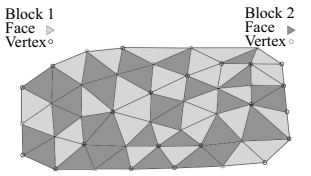
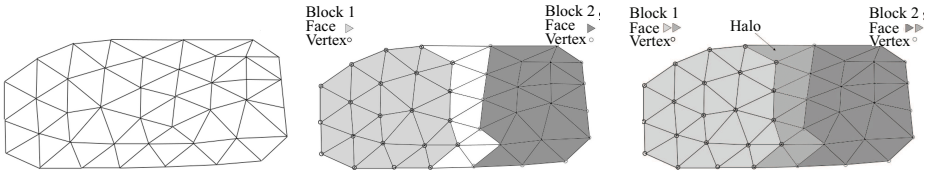


Fig. 5: Vertices and faces arbitrarily assigned to blocks

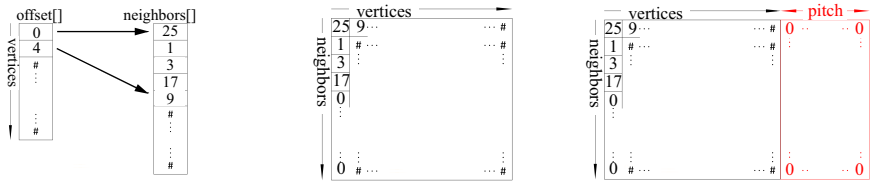
For *Fusion 3* we merge kernels *update()* and *compute()* to reduce data transfers for intermediate results. We use double-buffering to avoid data corruption due to concurrent access to function values f_n and f_c while computing the gradients involving *halo* faces. With all three kernel fusions we reduce the computational flow optimally and reorganize data to improve memory accesses as explained next.

Memory optimizations. There are two main areas of improving memory accesses: (a) The same function values are accessed multiple times during the kernel run and (b) some intermediate results reside in global memory. We first change array indices to point to a fixed size shared memory that stores face-gradients for a given block. Next, we use *shared memory* for vertex-gradients which is



(a) Input mesh (b) Inner vertices + faces (c) Partitioned blocks

Fig. 6: Partitioning a mesh into two blocks



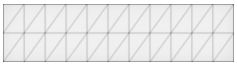
(a) CSR representation (b) 2D array (c) Pitched 2D array

Fig. 7: Reorganizing neighborhood references using *device arrays*

slightly complicated. With this new optimization, each thread first calculates the face-gradients and then adds the result to the vertex-gradient stored in the shared memory. Thus, to avoid data corruption due to concurrent accesses during vertex-gradient computations, we make some operations atomic. Using shared memory in conjunction with kernel fusions yields, in general, a multiplicative gain of 2.25× over our *baseline* version.

Next, we use *device arrays* to make memory accesses more efficient. Efficient in this context means coalescing loads and stores. Global memory is divided into segments of 128 bytes. Thus, coalescing global memory accesses needs manual reorganization of the memory. 2D and 3D device arrays support such memory reorganizations. In the following, the process for using device arrays is explained in a simplified manner using the Laplacian computation as an example. Now, the neighbors for each vertex are stored using compressed sparse row representation (CSR), refer Figure 7a. In each iteration, all threads of a warp access the neighbor with the same neighbor count. This leads to an inefficient access pattern since the array is sorted by vertex and not by neighbor count. Moreover, the first thread of any warp except the first warp accesses a completely random address. For these memory transactions to be fully coalesced two things need to be fulfilled. (1) The first thread of any warps needs to access the first element of a 128 byte segment. (2) All following threads need to access consecutive elements. For this purpose we store the neighbors in a 2D array as shown in Figure 7b. The vertices are stored into columns where column i stores the neighbors for vertex i. Therefore each row stores the neighbors with the same neighbor count. This representation thus fulfills the second requirement for full coalescing since the 2D array is stored to memory by rows. Next, we add a pitch at the end of each row to ensure that the first entry for the next row begins at an index that is a multiple of 32, see Figure 7c. This thus satisfies our first criterion for full coalescing. Note however, that 2D device arrays are not sufficient to fully coalesce this access in our case. 3D arrays are needed where the third dimension is the block count. Each slice then represents one block.

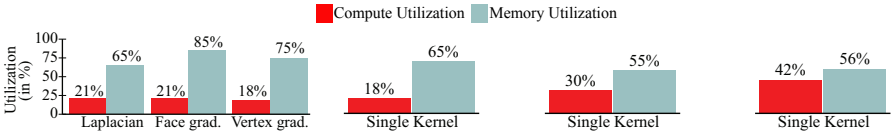
Using device arrays in conjunction with previous optimizations yields, in general, a multiplicative gain of 4× in comparison with our *baseline* version. We tried other optimization techniques such as using *vector types*. However, the performance gains for those techniques is not significant and we omit them from our discussion.



Multiprocessors	**13**
Cores/MP	**192**
CUDA Cores	**2496**
Threads per MP	(max) **2048**
Blocks per MP	(max) **16**
Threads per block	(max) **1024**
Global memory	**4800 MB**
Shared memory per MP	**48 KB**
Registers per MP	(32 bit) **64K**

(a) A rectangular mesh (b) *Kaa* the virtual lab snake (c) GPU platform

Fig. 8: Test data and GPU platform details



(a) *Fusion 1* (b) *Fusion 3* (c) *Shared memory* (d) *Device arrays*

Fig. 9: Compute and memory unit usage for various optimization steps

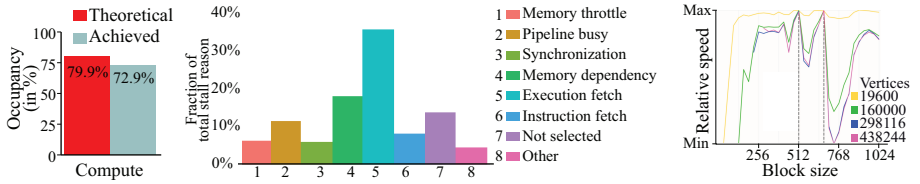
5 Results and Performance analysis

In this section we provide an in-depth performance analysis of our implementation with various optimizations. We use an Intel Xeon E5620 processor platform with 4(×2) cores @2.4 GHz for a naive, multicore reference CPU implementation.[1] Our GPU optimized implementations run on an NVIDIA Tesla K20c platform @704MHz (core) and @2.6GHz (memory), see Figure 8c for hardware details. For evaluations we use a set of 50 highly regularrectangular meshes ranging in resolution from 20K to 1M vertices, see Figure 8a. We also use a virtual lab snake (*Kaa*) with arbitrary surface geometry as shown Figure 8b. *Kaa* has 47867 vertices and 94088 triangular faces and a high irregularity in connectivity (neighborhood variance of 3.6 vertices). We also use NVIDIA's graphical profiling tool that offers a collection of different metrics to analyze GPU-accelerated applications.

Performance Evaluation. We first present *utilization metrics* profiled for our different optimization steps using our rectangular mesh with 1M vertices over individual kernel executions. Ideally one would expect 100% utilization for both *compute units* and *memory bandwidth*, but a benchmark of 60% simultaneous utilization is often considered very good in practice.

Figure 9a depicts utilization metrics with a barchart for the three main computing kernels for our optimizations in *Fusion 1*. An average compute unit utilization of around 20% implies that the GPU cores spend most of their time waiting for the data to arrive. Figures 9b, 9c, and 9d show that we progressively improve the compute unit utilization with each additional optimization step. For

[1] Our CPU implementation is not explicitly optimized for the said platform.



(a) Occupancy (b) Stall profile (c) Block-size vs. speed

Fig. 10: Various performance profiles: (a) Occupancy metrics for the *Fusion 2* step with the best launch configuration, (b) relative stall profile for the final, most optimized implementation, and (c) impact of the block-size in the launch profile on performance

our final optimized implementation (Figure 9d) we have achieved a considerable improvement at 42% compute unit utilization while compromising marginally on the memory bandwidth utilization at 56%. Importantly, using *device arrays* increased global load efficiency from 34.6% to 88.9% (not shown in the charts) and decreased global load transactions by over 60% (not shown). Thus, for a data intensive application with relatively limited computational operations, this is a very good trade-off in terms of utilization of the compute and the memory units.

Next, we present an examination of the occupancy metrics. Figure 10a shows the theoretical and achieved occupancies for the best launch configuration for our *Fusion 2* implementation. A considerably low theoretical occupancy (79.9%) in this case is compounded with even lower achieved occupancy (72.9%). This simply means that increasing occupancy with a different configuration does not increase performance and indicates that Fusion 2 has latency issues. Some of these issues are resolved with shared memory and especially device array. Our final optimized implementation has theoretical occupancy of 98.4% and achieved occupancy of 81.5% for single-precision and 78.2% for double-precision operations.

We also examine the system latencies for our final optimized implementation with single-precision operations. Figure 10b shows the breakdown of stall reasons averaged over an entire kernel execution. The largest share for warp latencies are due to execution dependencies (34.7%) and memory dependencies (17.3%). Memory dependencies are addressed by improving data access patterns. At this point the only inefficient global memory accesses are those to the function value arrays. There are a few common techniques to deal with irregular patterns such as use of: (a) L1 Cache, (b) texture memory, and (c) increased use of shared memory. We tried these options with insignificant or no performance gains. Also, execution dependencies can be mitigated with the use of instruction-level parallelism (IPL). In our case this doesn't work either due to: (a) shared memory overuse, (b) register overuse, or (c) thread overburden. Thus the stall profile show that any further gains in performance may only be achieved with severely diminishing returns on significant programming efforts.

For optimal performance we need to choose a suitable launch configuration, which we found in an empirical manner. Figure 10c shows plots for relative speeds

Name	Speed up factor			Techniques used
	R1: 19,600	R2: 1,000,000	Kaa	
Baseline	101	128	98	-
Fusion 1	151 (1.5)	189 (1.47)	122 (1.25)	Kernel Fusion
Fusion 2	148 (0.98)	173 (0.92)	125 (1.02)	Kernel Fusion, Partitioning
Fusion 3	160 (1.08)	182 (1.05)	132 (1.06)	Kernel Fusion, Double-Buffer
Shared	234 (1.47)	293 (1.61)	201 (1.52)	Shared Memory, Atomic Operations
Device	366 (1.56)	506 (1.73)	275 (1.37)	Device Array (2D and 3D)
Vector	379 (1.04)	535 (1.06)	317 (1.15)	Vector Type
Double	269 (0.71)	357 (0.67)	224 (0.71)	Double-Precision

Fig. 11: Speed up factor for each optimization with respect to a naive multicore CPU and with respect to the previous optimization (in brackets). **R1** and **R2** are rectangular meshes of specified resolutions

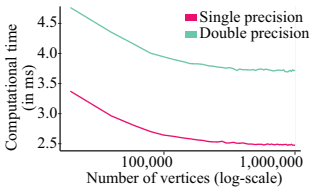


Fig. 12: Performance comparison of single and double-precision. Time per iteration per 1000 vertices of the mesh

versus block-size for a given mesh resolution. The block-size is the most important factor in determining a launch configuration and for most of the resolutions a block-size of 512 or 672 gave best performance. We found that choosing a target block-size of 512 with an assumption of 20% halo faces yields near optimal launch configuration (i.e., atleast 99% of the best configuration performance), in general.

Speedups. Finally, we summarize the performance gains for our various optimizations in Figure 11. We also plot the speed of our final optimization in absolute time in Figure 12 for the single and double-precision implementations for a rectangular mesh with different resolutions.

6 Conclusions

We have described an optimized GPU implementation for solving non-linear reaction-diffusion (RD) PDEs, which play a crucial role in biological pattern formation, on triangulated meshes in 3D. Optimized tools like our approach will be useful for biologists who study RD models in realistic scenarios with 3D geometries, since they allow them to explore the parameter space of these systems more efficiently. Our works shows that with careful optimizations that take into account parallelism, locality, and recomputation, we can gain a significant speedup over a naive GPU implementation. Therefore, we believe it would be fruitful to allow programmers to explore these degrees of freedom more easily, without resorting to low-level CUDA implementation as we did. Unfortunately, current domain-specific languages for mesh based PDEs do not support this type of optimization. While the Halide language [12] successfully allows the programmer to optimize parallel code execution at an abstract level, and separately from algorithm specification, this approach is tightly connected to the rectangular topology of images (or higher dimensional regular grids). In particular, it exploits the possibility to specify the order in which the dimensions of the grid should be traversed, and whether they should be traversed sequentially or in parallel. It is an interesting challenge for future work to generalize these concepts to arbitrary meshes, such that they can easily be exploited by a programmer for code optimization by providing high-level specifications.

References

1. Tracking bifurcating solutions of a model biological pattern generator. {IMPACT} of Computing in Science and Engineering 2(4), 355 – 371 (1990)
2. A framework approach for developing parallel adaptive multiphysics applications. Finite Elements in Analysis and Design 40(12), 1599 – 1617 (2004), the Fifteenth Annual Robert J. Melosh Competition
3. Balay, S., Gropp, W., McInnes, L., Smith, B.: Efficient management of parallelism in object-oriented numerical software libraries. In: Arge, E., Bruaset, A., Langtangen, H. (eds.) Modern Software Tools for Scientific Computing, pp. 163–202. Birkhäuser Boston (1997)
4. Brandvik, T., Pullan, G.: Sblock: A framework for efficient stencil-based pde solvers on multi-core platforms. In: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology. pp. 1181–1188. CIT '10, IEEE Computer Society, Washington, DC, USA (2010)
5. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based pde solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 9:1–9:12. SC '11, ACM, New York, NY, USA (2011)
6. Giles, M.B., Mudalige, G.R., Sharif, Z., Markall, G., Kelly, P.H.: Performance analysis of the op2 framework on many-core architectures. SIGMETRICS Perform. Eval. Rev. 38(4), 9–15 (Mar 2011)
7. Karypis, G.: Metis - serial graph partitioning and fill-reducing matrix ordering (march 2013), http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
8. Kondo, S., Arai, R.: A reaction-diffusion wave on the skin of the marine angelfish pomacanthus. Nature 678(376), 765 – 768 (1995)
9. Kondo, S., Miura, T.: Reaction-diffusion model as a framework for understanding biological pattern formation. Science 329(5999), 1616–1620 (2010)
10. Meyer, M., Desbrun, M., Schröder, P., Barr, A.H.: Discrete differential-geometry operators for triangulated 2-manifolds. In: Visualization and mathematics III, pp. 35–57. Springer (2003)
11. Murray, J., Myerscough, M.: Pigmentation pattern formation on snakes. Journal of Theoretical Biology 149(3), 339 – 360 (1991)
12. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. 31(4), 32:1–32:12 (Jul 2012)
13. Reuter, M., Biasotti, S., Giorgi, D., Patan, G., Spagnuolo, M.: Discrete laplace-beltrami operators for shape analysis and segmentation. Computers and Graphics 33(3), 381 – 390 (2009), {IEEE} International Conference on Shape Modelling and Applications 2009
14. Turing, A.M.: The chemical basis of morphogenesis. Philosophical Transactions of the Royal Society of London B: Biological Sciences 237(641), 37–72 (1952)