We discuss the implementation of a Runge Kutta Discontinuous Galerkin (RKDG) scheme for solving the two-dimensional depth-averaged shallow water equations on a graphical processing unit (GPU) using the CUDA API. Due to the compact formulation of the scheme it is well suited for parallelization and hardware acceleration by general purpose GPUs. The performance of the code is compared twice, on the one hand against a standard serial code, and on the other hand against parallelization on multi-core CPUs. Furthermore, we discuss the speed-up benefit from the point of view of practical engineering applications and compare it against the effort of conducting the code parallelization.

INTRODUCTION

Standard Galerkin finite element methods provide unsatisfactory results for advection-dominated flows. Since continuous finite elements imply energy conservation, the numerical solution shows oscillations in advection-dominated parts of the solution and at shocks [1]. One breakthrough was achieved by the introduction of a streamline upwind (SU) weighted test function by Hughes & Brooks [6] and its application to shallow water flows by Katopodes [7]. The upwinded test function produces a sufficient amount of dissipation to dampen the oscillations and decrease the energy head at jumps. A major difficulty of SU methods is to restrict the use of this dissipation to non-smooth regions of the solution, in order to prevent decreases in accuracy in the smooth regions. Most SU methods use implicit time integration. An explicit formulation seems to be useful only for very unsteady flows requiring a CFL number close to 1, in order to obtain sufficient accuracy in time [8]. However, the explicit application of the SU method seems to be inefficient compared to that of the finite volume methods because the latter require the solution of a linear equation system at every time step for decoupling the mass matrix.

In this paper, we present a scheme based on a Discontinuous Galerkin (DG) finite element method for spatial discretization [3]. The discrete variables on the nodes of the
inter-cell boundaries are shared by two elements. For a discontinuous discretization each
element has its own discrete variable at the inter-cell boundary forming a discontinuity. A
main advantage of the DG method over continuous finite element methods with explicit
time integration is the completely decoupled mass matrix obtained by using orthogonal
shape functions. This avoids the solution of linear equation systems to invert the mass
matrix while using an explicit Runge Kutta (RK) time integration. Furthermore, the
computation of a numerical flux at the inter-cell boundaries introduces upwinding into
the scheme while keeping the Galerkin test function in the element. The DG method was
originally implemented to the shallow water equations (SWE) by Schwanenberg &
applications to inundation modeling and dam-break flows. Further examples are shown in
Ghostine et al. [5], Bunya et al. [2] and Ern et al. [4].

Recently, the developments in computer hardware move away from increasing clock
speed towards providing many computational cores that operate in parallel. This is true
for general purpose central processing units (CPUs), but in particular for general purpose
graphics processing units (GPGPUs). Recent work on the parallelization of the DG
scheme for Maxwell’s equations for GPGPUs [9] show the computational potential of the
scheme on massive parallel hardware. Keeping in mind the time intensive computations
for large scale inundation and dam-break simulations, we investigate the parallel
efficiency of the DG method in application to the SWE. Focusing on typical engineering
applications applied by small and medium-size consultants, we concentrate on computers
with multi-core CPUs and / or additional GPGPUs, and neglect computational clusters.
This limits the parallelization methods to OpenMP and special toolkits for accelerator
cards such as the CUDA API. Typical techniques for clusters such as domain
decomposition with message passing are not considered in our research. The reader is
referred to Neal et al. [10] for a more general overview about parallelization techniques
in inundation modeling.

HYDRAULIC MODEL

The mathematical description of a depth-averaged incompressible fluid is given by the
2D depth-averaged shallow water equations in the form

\[ \partial_t U + \nabla \cdot (F(U), G(U)) = S(U) \]  \hspace{1cm} (1)

where
\[ \mathbf{U} = \begin{pmatrix} h \\ p \\ q \end{pmatrix}, \quad \mathbf{F} (\mathbf{U}) = \begin{pmatrix} p \\ \frac{p^2}{h} + \frac{1}{2} gh^2 \\ \frac{pq}{h} \end{pmatrix}, \quad \mathbf{G} (\mathbf{U}) = \begin{pmatrix} q \\ \frac{pq}{h} \\ q^2 + \frac{1}{2} gh^2 \end{pmatrix}, \] (2)

\[ S (\mathbf{U}) = \begin{pmatrix} 0 \\ gh (\partial_z z_b - \frac{n^2 p \sqrt{p^2 + q^2}}{h^{10/3}}) \\ gh (\partial_z z_b - \frac{n^2 q \sqrt{p^2 + q^2}}{h^{10/3}}) \end{pmatrix} \] (3)

in which \( h \) = water depth; \( p, q \) = unit discharge in \( x \) and \( y \)-directions; \( g \) = acceleration due to gravity; \( \rho \) = fluid density; and \( z_b \) = channel bed elevation. Bed roughness is approximated by the empirical Manning formula with Manning’s roughness coefficient \( n \).

The weak finite-element formulation of the equations is given by

\[
\int_K \mathbf{U}_h \mathbf{v}_h \, dK + \int_{\partial K} (\mathbf{F} n_x + \mathbf{G} n_y) \mathbf{v}_h \, d\Gamma - \int_K (\mathbf{F}, \mathbf{G}) \cdot \nabla \mathbf{v}_h \, dK = \int_K \mathbf{S} \mathbf{v}_h \, dK
\] (4)

Here, \( n_K = (n_x, n_y)^T \) is the unit vector outward normal to the element boundary \( \Gamma \) of the finite-element \( K \), \( \mathbf{v}_h \) is an arbitrary smooth test function, and \( \mathbf{U}_h \) is the approximation of \( \mathbf{U} \). Since the RKDG method allows for variables at the element edges to be discontinuous, the inter-cell flux depends on the values \( \mathbf{U}_h \) in each of the two elements. It is therefore not uniquely defined and is replaced by a numerical flux \( H \). The approximated variable \( \mathbf{U}_h \) can be expressed as a sum over the products of discrete variables \( U_l \) and shape functions \( \phi_i \).

\[
\mathbf{U}_h (x, y) = \sum_l U_l \phi_l (x, y)
\] (5)

Using the standard Galerkin approach, shape functions and test functions are identical. Rewriting equation (4), the DG space discretization can be summarized by a system of ordinary differential equations as

\[
\frac{d}{dt} \mathbf{M} \mathbf{U}_h = \mathbf{L}_h (\mathbf{U}_h), \tag{6}
\]

where \( \mathbf{M} \) is the mass matrix and \( \mathbf{L}_h \) is an operator describing the spatial discretization. By using orthogonal shape functions, the mass matrices \( \mathbf{M} \) in the linear equation system (6)
become diagonal. For triangular elements, a linear shape function can be obtained by choosing a function that takes the value of 1 at the mid-point \( m_i \) of the \( i \)-th edge of a triangle, and the value of 0 at the midpoints of the other two edges. Thus, the diagonal mass matrix can be expressed as follows:

\[
M = |K| \text{diag} \left( \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right)
\]

The integrals are approximated by a three mid-point rule for the triangles and a two-point Gauss integration for the line integrals.

The equations (6) are integrated in time as follows by using the following second-order two-step TVD Runge Kutta method for a linear space discretization,

\[
U_h^1 = U_h^n + \Delta t \mathbf{L}_h(U_h^n), \quad U_h^{n+1} = \frac{1}{2} U_h^n + \frac{1}{2} U_h^1 + \frac{1}{2} \Delta t \mathbf{L}_h(U_h^1)
\]

A slope limiter is applied on each step of the Runge Kutta scheme for achieving TVD properties of the scheme, for details see Schwanenberg & Harms [14].

**PARALLEL IMPLEMENTATION**

The coding of the RKDG method is straightforward, following the set of equations defined above. Due to the orthogonal shape functions and the explicit Runge Kutta time stepping, the code becomes a sequence of loops on edges for computing the numerical fluxes and elements for computing the spatial discretization, updating the states, and applying the slope limiter (Figure 1).

```plaintext
1  time loop {
2      element loop {
3          check for minimum time step;
4      }
5  }
6  apply boundary condition;  % 1st Runge Kutta step
7  edge loop {
8      compute numerical flux \( \mathbf{H} \);
9  }
10  element loop {
11      compute spatial discretization \( \mathbf{L}_h \), compute \( U_1 \);
12  }
13  element loop {
14      apply slope limiter on \( U_1 \);
15  }
16  apply boundary condition;  % 2nd Runge Kutta step
17  ...  
18  }

Figure 1  Pseudo code for RKDG code with two-step Runge Kutta time stepping (parallelized loops are bold)
```
From our point of view, the code parallelization should meet the following two main demands: i) keep as possible the same code both for serial and parallel code versions of the solver aiming at simple maintenance, ii) minimize the work input for parallelization. Therefore, we first of all rely on pragma-based optimization techniques such as OpenMP for parallelizing the code for multi-core CPUs. The same concept was originally intended for the GPU using PGI-compiler specific pragma-directives. However, the approach didn’t work to our full satisfaction related to indirect addressing and the incapacity of the compiler to identify the corresponding allocated memory for copying it from CPU to GPU and vice versa. Therefore, we applied the CUDA API which resulted in small code changes, mainly applied to the administration of the parallel loops.

Having a look at the pseudo code in more detail, the element loop in line 3 is not parallelized for the time being, because of being responsible for only a small percentage of CPU time less than 0.3% of the total execution time. Furthermore, the execution time for the assignment of boundary conditions in line 7 is proportional to the square root of the number of elements and therefore also not parallelized. The edge loop in line 8 is parallelized. It includes some indirect addressing for reading element data into the edges and writes numerical fluxes back into the elements. The next element loop in line 11 is completely local. The limiter loop in line 15 includes read-only access to mean values of neighboring elements, and is local otherwise.

Profiling of the application on a single core shows that the proportion \( P \) of the program made parallel by the steps above is larger than 99.5%. We use Amdahl’s law for estimating the maximum parallel speedup that can be achieved by using \( N \) processors

\[
speedup = \frac{1}{(1 - P) + \frac{P}{N}},
\]

and get a theoretical speedup of 200 for an infinite number of processors. Taking into account the 240 processing units available on a Tesla C-1060 GPU we get a maximum speedup of about 110.

Furthermore, the parallel efficiency is defined as the quotient of speedup and \( N \). This analysis neglects other potential bottlenecks such as memory bandwidth and I/O bandwidth, if they do not scale with the number of processors.

RESULTS

Performance tests on the parallel speedup of the solver are performed on a number of typical hardware environments (Table 1). The first one is a typical notebook, worth about € 800, being used by the first author for simulations when being abroad. The second system represents a common server with two Quad Core XEON CPUs, representing a value of about € 6000. The third and last system is a dedicated computer for performing numerical simulation (€ 1500) including a TESLA C-1060 GPU for about € 1000.
We selected three test cases in order to get a representative picture of the parallel performance of the solver (Table 2). The first one is a typical riverine application with a fully wet domain. The second one is a dam-break application, in which most of the domain is dry in the beginning, being flooded partially in the course of the computation. The test is included to check the ability of the code for load balancing within the loops taking into account that dry elements require significantly less computational effort. Test case 3 is an academic exercise with a square domain of different sizes for checking the scaling of the solver due to different numbers of elements.

<table>
<thead>
<tr>
<th>Test case</th>
<th>execution time / (time step * element) [micro s], parallel efficiency [-]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3.062</td>
</tr>
<tr>
<td></td>
<td>1.721</td>
</tr>
<tr>
<td>2</td>
<td>2.263</td>
</tr>
<tr>
<td></td>
<td>1.380</td>
</tr>
<tr>
<td></td>
<td>0.876</td>
</tr>
<tr>
<td></td>
<td>0.571</td>
</tr>
<tr>
<td>3</td>
<td>3.705</td>
</tr>
<tr>
<td></td>
<td>1.952</td>
</tr>
<tr>
<td></td>
<td>1.081</td>
</tr>
<tr>
<td></td>
<td>0.712</td>
</tr>
<tr>
<td></td>
<td>0.203</td>
</tr>
</tbody>
</table>

Table 1 Test environment (hardware and software framework)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>OS / Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Intel Core2 Duo T7300 @ 2.00GHz, 1.96 GB RAM</td>
<td>Windows XP (32-Bit)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Intel XEON 2x E5440 @ 2.83 GHz, 8.00 GB RAM</td>
<td>Windows 2003 Server (32-Bit)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Intel XEON 2x E5504 @ 2.00GHz, 8.00 GB RAM Nvidia Tesla C-1060</td>
<td>Ubuntu Linux 9.10 x86_64</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Test cases

<table>
<thead>
<tr>
<th>Number of points / number of elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 13821 / 27000</td>
<td>river section with groyne field, completely wet domain</td>
</tr>
<tr>
<td>2 13541 / 26041</td>
<td>Malpasset dam-break simulation according to Schwanenberg &amp; Harms [14], partially dry domain</td>
</tr>
<tr>
<td>3a 63001 / 125000</td>
<td>square area with 250m x 250m, 2 triangles per square meter, completely wet domain</td>
</tr>
<tr>
<td>3b 251001 / 500000</td>
<td>square area with 500m x 500m, otherwise like 3a</td>
</tr>
<tr>
<td>3c 1002001 / 2000000</td>
<td>square area with 1000m x 1000m, otherwise like 3a</td>
</tr>
</tbody>
</table>

Table 3 Execution times and parallel efficiency (preliminary projected GPU results based on CUDA code coverage of about 60% of total code)
The results of the tests for the combination of test environment and test case are summarized in Table 3. For comparison purposes, we provide the execution time per time step and element in micro seconds as well as the parallel efficiency for a different number of cores.

Not surprisingly, the theoretical parallel efficiency of slightly less than 1.0 on multi-core CPUs is not achieved in any of the test cases. The maximum parallel efficiency on multi-core CPUs with a value of 0.96 is achieved with 2 cores on the largest test case. The efficiency is decreasing with the number of cores to a level of about 0.5 - 0.8 for 8 cores depending on the size of the test case and the OS / compiler combination. To our best guess, this decrease is related to bottlenecks in the memory bandwidth in combination with indirect addressing. The efficiency of the test case with a partially dry domain is less than the one in the other test cases, but still on an acceptable level between 0.87 (best result on 2 cores) and 0.35 (worst result on 8 cores).

The execution time on the GPU is 5 – 21 times faster compared to that of single thread execution on the CPU of test system 3. The speedup is smaller by a factor of 1.7, if comparing it to the faster CPU of test system 2. Related to the 8 core execution on test system 2, the GPU is about 1.5 times faster for the small test problem 2. For the larger test problems 1, 3a-c the speedup is converging to 3.

Focusing on the larger test problems and taking into account hardware costs, the GPU is the most cost effective option for performing simulations. It is followed by the low budget simulation PC (3) and the notebook system (1) with a dual-core CPU. Using the product of execution time and costs as a performance measure, these systems lack behind by a factor of about 5 and 7, respectively. The worst cost efficiency is achieved by the server system (3) with 8 processing units and factors of about 18.

CONCLUSIONS

The code parallelization of the DG scheme turned out to be easy for OpenMP. It was finalized within a couple of days. The effort for the GPU parallelization using the CUDA API is moderate and took about one to two weeks, mainly related to some training on the language extension and optimization of memory allocation and administration.

The application of a dedicated GPU turns out to be the most cost effective option for performing large scale inundation and dam-break simulations. It is therefore recommended, if frequent or time critical simulations have to be performed. In other cases, the OpenMP code version offers an attractive option for exploiting available multi-core CPU resources.

ACKNOWLEDGEMENT

The authors thank Mike Walsh and NVIDIA Ltd for providing hardware for development and testing.
REFERENCES