High-Performance Simulations on GPUs using Adaptive Time Steps*

Marcel Köster*[1], Julian Groß[1], and Antonio Krüger[1]

Saarland Informatics Campus, Campus D3.2, 66123 Saabrücken, Germany
{marcel.koester,julian.gross,antonio.krueger}@dfki.de

Abstract. Graphics Processing Units (GPUs) are widely spread nowadays due to their parallel processing capabilities. Leveraging these hardware features is particularly important for computationally expensive tasks and workloads. Prominent use cases are optimization problems and simulations that can be parallelized and tuned for these architectures. In the general domain of simulations (numerical and discrete), the overall logic is split into several components that are executed one after another. These components need step-size information which determines the number of steps (e.g. the elapsed time) they have to perform. Small step sizes are often required to ensure a valid simulation result with respect to precision and constraint correctness. Unfortunately, they are often the main bottleneck of the simulation. In this paper, we introduce a new and generic way of realizing high-performance simulations with multiple components using adaptive time steps on GPUs. Our method relies on a code-analysis phase that resolves data dependencies between different components. This knowledge is used to generate specially-tuned execution kernels that encapsulate the underlying component logic. An evaluation on our simulation benchmarks shows that we are able to considerably improve runtime performance compared to prior work.

Keywords: Simulations, parallel computing, adaptive time steps, graphics processing units, GPUs

1 Introduction

There have been many distinct time-step adaption methods for completely different domain-specific problems. Such adaption techniques are particularly well investigated in the field of fluid and/or particle-based simulations. Prominent examples are SPH (Smoothed Particle Hydrodynamics) [10, 21, 22] and Position Based Dynamics (PBD) [23] simulations. Even sophisticated optimization problems are often modeled with the help of underlying rule-based and simulation like programs or code fragments [11–13]. Regardless of the use case and the domain, a simulation always consist of several phases that have to be evaluated one after another[1]. Executing those parts in an iterative manner yields the final

* PREPRINT

[1] A single simulation iteration is commonly referred to as a simulation step.

simulation result in the end [16,24]. An essentially required piece of information in this context is the actual time-step size which is used to execute each simulation step. Choosing this step size is very important with respect to simulation correctness and performance at the same time: Small time-step sizes often yield the best precision but are much more expensive since the simulation has to execute more steps until it reaches a final state. Since many parts of a simulation step have to be executed sequentially, the time-step size is often considered to be the primary performance-critical part.

Today, the mentioned simulations are often run on GPUs to benefit from their parallel computation capabilities. This allows to realize large-scale simulations which are even well suited for real-time applications [17,19]. In order to use these massive processing features, algorithms (as well as their surrounding applications) have to be specifically tuned for these architectures. From a high-level point-of-view, each simulation consists of a set of *components* which describe a specific module of the whole simulation logic.

In this paper, we present a new method to realize component-based simulations on GPUs using adaptive time steps. Our method is generic in respect of the application domain, as it does not rely on specific knowledge about the internal structure of the components. Instead, we use a static program analysis to determine a data-dependency graph. Based on this graph, we are able to compute the next adaptive step size. In this scope, we also allow developers to include their domain-specific knowledge with respect to time-step constraints. Our concept is based on the well-known idea to interpolate certain values at specific points in time [3]. We use this principle to simulate an intermediate component execution that has not happened in order to reduce the overall runtime of the simulation. Using our approach yields speedups between 12% (a smaller gravity-like simulation) and 22% (a larger PBD-like simulation) on our non-optimistic evaluation scenarios. Furthermore, we do not suffer from slowdowns in comparison to more conservative time-stepping approaches. This makes it a perfect extension for modern GPU-based simulation systems.

The remainder of this paper, summarizes and discusses related work from the fields of adaptive (fluid/particle) simulations to improve performance. We give a high-level introduction into the modeling of simulations in Section 3. This section also describes the major challenges in terms of adaptive time steps that we can solve using our new approach. Section 4 presents our generic concept and gives in-depth information about all design considerations and implementation details on GPUs. The evaluation section covers two designed benchmark scenarios. They are inspired by real-world simulation models to measure realistic performance numbers.

2 Related Work

From a theoretical point-of-view, using interpolation functions to adapt time steps is a well known concept [2, 3, 9, 27]. There have been many different approaches from the field of numerics (solving partial differential equations, for example). They also leverage interpolation functions to resolve intermediate values. This contribution has a different view on adaptive time steps without having

explicit domain knowledge about the components. As outlined above, we focus on practical and pragmatic aspects of realizing fast and efficient simulations (see Section 5). To the best of our knowledge, we consider the following papers to be related to this subject.

Adams et al. [1] present an adaption approach for particle-based fluids. They define a domain-specific criterion to decide which region of the fluid simulation is more important and needs more computational resources. In contrast to our method, they are not adapting the actual time-step sizes, but focus on adaptive particle sizes to reduce the computational complexity.

Predictive-Corrective Incompressible SPH (PCISPH) [29] is a well known fluid-computation model that essentially predicts forces to enable larger time steps in order to overcome the severe time-step restrictions of previous papers. Ihmsen et al. [8] add support for adaptively computed time steps in the context of PCISPH. They use domain-specific properties of the underlying PCISPH algorithm and combine them with knowledge about maximum time-step restrictions of these simulations.

Further adaptive fluid simulations are the ones by Hong et al. [6] and Zhang et al. [30]. They basically split and merge particles in order to reduce the number of elements to process. The split- and merge-criteria are based on a set of scenario-dependent properties. It is regrettable that these methods essentially modify the underlying data to be evaluated by several components. This requires domain-specific knowledge about the underlying structure of the problem and its associated optimization possibilities which cannot be easily generalized to work with black-box components.

Solenthaler et al. [28] and Horvath et al. [7] use a custom solution to enable adaptive computations in the scope of fluid simulations. They are achieving this purpose by coupling a set of differently scaled simulations. This avoids splitting and merging of particles. As before, these methods are limited to their particular field of application and cannot be applied to generic components.

Macklin et al. [18] use a PBD-based algorithm and add support for density constraints in order to model fluids. Since PBD simulations are very robust, they relax time-step restrictions of previous approaches significantly. Koester et al. [14] present an adaption scheme for these simulations. Like Macklin et al. [18], they are using an iterative constraint solver to move particles into the right positions to satisfy a fluid-density constraint. During solving, some particles can be considered less important than others. The more important a particle is, the more particle-position adjustments will be performed in upcoming iterations of the constraint solver. This significantly reduces the overall computational effort, since less important particles will not be considered in further solver iterations. This idea is similar to our high-level concept of interpolated values at certain time steps: Some values loaded from memory are not available at particular points in time and can be reconstructed using interpolation functions. In contrast to the original paper by Koester et al., we can provide intermediate values at specific time steps, whereas the related algorithm uses out-dated information without interpolation.

Garcia et al. [4] describe an adaptive time-stepping method that computes common step sizes using global reduction kernels. They estimate the time-step

size in each iteration and synchronize the resolved time-step value with the CPU part of the application. We follow their approach by applying all time-step estimation methods of all components prior to the execution part of each component: We still rely on a synchronization step with the CPU side to exchange information about the computed time-step size. In addition, our method requires further logical steps to compute a common step size across all components (see Section 4).

Mayr et al. [20] leverage an iterative computation of the time-step size based on certain error calculations in the field of fluid-interaction solvers. From a high level point-of-view, this is closely related to our method: We compute the time-step size for each component and search for a compatible step size that is acceptable for all components. However, their adaption method relies on specific domain knowledge (which is tightly coupled to their application domain) that cannot be reused in a generic and portable way.

3 Component-Based Simulations on GPUs

This section covers a brief introduction into the general modeling of simulations on GPUs consisting of several components C_0 to C_{n-1}. Every component realizes a certain part of a simulation loop and is supposed to compute a separate piece of the overall puzzle that we want to put together. Figure 1 shows a sample simulation loop consisting of several components. Note that the overall execution order is usually determined by the domain expert/application programmer that has to take all data dependencies into account. Moreover, many simulations leverage the concept of *double buffering* to simplify data dependencies. Double buffering allows to read the same source data in each component while writing simulation updates into a separate target buffer. This ensures that each component is able to work on a consistent state without having to worry about possible changes by other components. Alternatively, some applications work on a single buffer only. This exposes all value updates to the components executed afterwards. The developers have to take special care to create a consistent logical component model that is compliant with the actual buffering approach used.

Fig. 1. A simple simulation loop with five components C_0 to C_4 in a sequential order. The back edge from C_4 to C_0 models the jump to the initial component C_0 of the next simulation step. Furthermore, it visualizes the data dependency between two steps.

Each component is applied to a set of items (e.g. particles or other data elements) from a set of source buffers. A *state* in memory thereby consists of all items in all buffers that the components can operate on. To implement this functionality on a GPU, each component C needs to have a separate GPU kernel that iterates over all items (referred to as the *range* of C). Algorithm 1 shows a straight-forward kernel implementation for an arbitrary C. It uses grid-stride loops to realize an efficient way to iterate over all items in the range of C. However, depending on the computational complexity and the performance characteristics of C, it can be beneficial to apply loop unrolling at this point [15].

Algorithm 1: Simple application kernel algorithm for component C

```
   /* Perform a grid-stride loop over the padded value range of C   */
1  for i := global index; i < range(C); i += grid size * group size do
   |  /* Initialize component by loading relevant information       */
2  |  c := C.Init(i);
   |  /* Evaluate component                                         */
3  |  c.Evaluate(Δt);
4  end
```

Applying components in the presence of adaptive time steps means that need to retrieve information about the potential step size of each component. The estimated step sizes also include domain-specific time-step constraints to satisfy correctness/stability requirements. Figure 2 shows the high-level difference between a workflow using fixed vs. adaptively computed time steps. In the adaptive setting, all components estimate their intended time-step size they could potentially, assuming that all necessary information is available (optimistic assumption). Afterwards, the minimum time step of all potential time steps (from the first phase) can be used safely (sound assumption). Finally, the computed step size from the previous phase can be used for all components in the actual simulation step. Consequently, we need to create additional kernels that compute the possible number of steps to execute all components.

Although this approach is perfectly sound with respect to time-step constraints, it does not solve all performance issues. The time-step computation kernels require an additional iteration over all items in the state which causes an overhead compared to the fixed-step version. The latter can only be outperformed by the adaptive one if it is possible to skip many simulations steps. From now on, we consider a step size of 1 as the reference and default fixed step size that satisfies all constraints. Since we have to use the minimum common step size, the overall probability that we have to perform a simulation step of size 1 is $P = 1 - p^n$, where p is the probability that a single component needs to perform a step of 1 and n is the number of components. Even in small examples, P can easily become close to 1. This in turn can also lead to a performance degradation (see Section 5).

4 Our Method

Figure 3 visualizes our high-level concept to determine intermediate item values. Formally, we apply a component C_i ($i \in [0, \ldots, \mathit{num\ components} - 1]$) to the current state S. More precisely, the component works on a part of the state S_i. Taking the current simulation time T and the next time step of size Δt into account yields a new state at time step $T + \Delta t$ that is given by

$$S_i(T + \Delta t) = C_i\left(S_i(T), \Delta t\right). \tag{1}$$

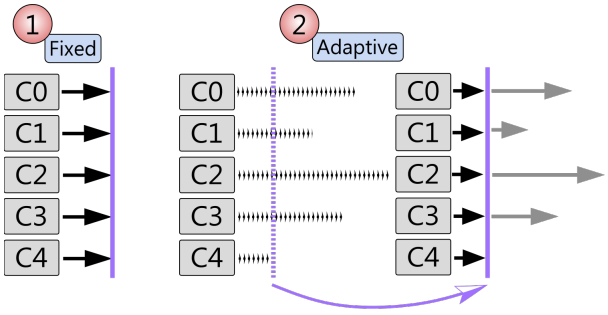

Fig. 2. A visualization of five components C_0 to C_4 and their executed step sizes (black arrows). A fixed step size that is uniform across all components (left, 1). A two-phase approach to compute a compatible step size (the minimum of all possible step sizes) across all components (right, 2). Grey arrows indicate the intended step sizes which could not be used for an actual simulation step.

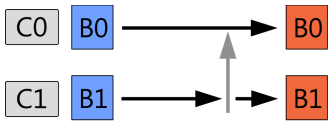

Fig. 3. Two components C_0 and C_1, where C_1 has a data dependency to the items in B_0. C_0 performs a large time step while reading from its source buffer and writing into its target buffer B_0. Our idea is to enable C_1 to do the same large time step, even if its dependencies only allow two small steps. This requires an interpolation of the values in B_0 (gray arrow) at the intermediate time step of C_1.

We then use a problem- and domain-specific interpolation function I (see Section 5) to approximate the state S at time $T + \Delta k$ in the future using

$$S_i(T + \Delta k) \approx I_{\Delta k}\left(S_i(T), S_i(T + \Delta t)\right), \Delta k \in [0, \ldots \Delta t], \tag{2}$$

where Δk is a limited time offset that has to be smaller or equal to Δt. Consider a case in which a component C_j accesses interpolated information provided by another component C_i, where $i \neq j$ (for example in Figure 3). Further, let us assume that we have to perform two small steps $\Delta l > 0$ and $\Delta o > 0$ that sum up to the intended step size $\Delta t = \Delta l + \Delta o$. This can be formally expressed via

$$S_j(T + \Delta l) = C_j\left(S_i(T), S_j(T), \Delta l\right), \tag{3}$$

$$\begin{aligned} S_j(T + \Delta t) &= C_j\left(S_i(T + \Delta l), S_j(T + \Delta l), \Delta o\right) && (4)\\ &= C_j\left(I_{\Delta l}\left(S_i(T), S_i(T + \Delta t)\right), S_j(T + \Delta l), \Delta o\right). && (5) \end{aligned}$$

These equations demonstrate that a single use of an interpolation function can cause a simulation deviation that easily propagates to all other components. However, potential differences in terms of the simulation results depend on a huge variety of different factors. For instance, the actual interpolation being used, the number of components that can access interpolated information and even the application scenario. In comparison to related approaches, this is not a novel limitation that only applies to our method. Other methods rely on domain-specific time-step computations that also introduce simulation deviations. We

support domain- and even scenario-specific time-step computations to model the required knowledge in the scope of our method (see also Section 4.2).

In order to apply interpolation properly to the right buffers, we have to identify locations that allow us to interpolate between items in the source and items in the target buffer. Therefore, we conceptually always rely on the idea of double buffering. However, if we execute a component an even number of times, while others have been executed an odd number of times, the items in the source and target buffers are mixed with respect to the different iteration steps. This issue is visualized in Figure 4: Since component C_1 is executed an even number of times, its originally used source buffer contains the finally written information. To circumvent this problem, C_1's kernel has to copy the contents of the memory buffers into its intended target buffer to make C_1's updates visible to all other components. Unfortunately, the problem is even worse: If another component needs to interpolate the values written by C_1, we need its original source-buffer data. Consequently, we have to copy all items that can potentially be used for interpolation into a separate memory buffer at the beginning of the simulation step.

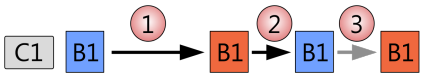

Fig. 4. Component C_1 from Figure 3 performing two small steps in order to reach C_0's larger time step. First, C_1 performs its initial step and writes its computed items into its associated target buffer (1). Afterwards, C_1 is applied again and performs the next step. In this case, C_1 reads its source values from the originally intended target buffer and writes into its original source buffer (2). To ensure that the finally written values will end up in the correct target buffer, we have to copy the updated items from the source into the target after execution of the second step (3).

Finding locations to safely apply interpolation to already computed items is based on a static program analysis. It essentially resolves a data-dependency graph induced by view accesses in the program (see Figure 5). Note that the decision on the back-edge dependency (or in other words: the first component in the schedule) is usually done by a domain expert. He or she is conceptually able to split the resolved dependency graph into semantically separate simulation steps. Although it is possible to use topological sorting of this graph to determine an execution order, it can happen that we encounter multiple possibilities to schedule a component. These cases require domain knowledge to decide on the actual component order.

Once the actual schedule is available and the component dependencies have been resolved, we can focus on the adaptive time-step size computation (see Figure 6). First, we perform a step estimation by querying all components (using *ComputeNumSteps* from Listing 1.1) that depend on immediate buffer information from a previous iteration (components that are reachable via back edges). In the case of multiple components that are reachable by following all back edges, we have to compute the minimum step size of all of them. Based on the actual domain, our approach is applied to, it may be totally fine to use the maximum possible time-step size from all components without explicit data dependencies.

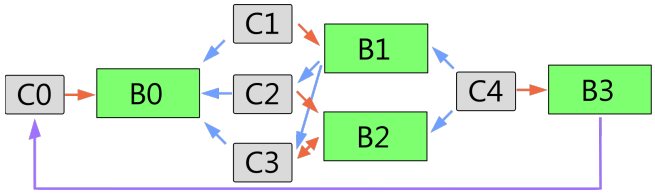

Fig. 5. A set of components C_0 to C_4 (from Figure 1) along with some imaginary read and write dependencies to intermediate buffers B_0 to B_3. The read dependency of component C_0 from buffer B_3 is highlighted in purple, since it is also implicitly given by the back edge of the simulation loop. Note that it is possible to automatically compute a component schedule by applying topological sorting. In this example, this could yield the schedule $C0, C1, C2, C3, C4$. Note further that the back-edge dependency of C_0 to buffer B_3 separates multiple simulation steps from each other, since C_0 is the first component is this schedule.

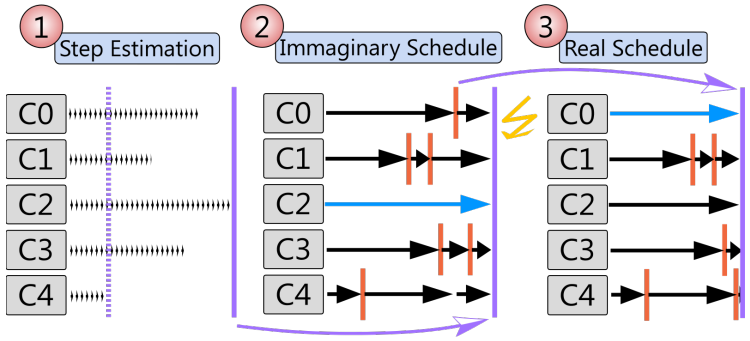

Fig. 6. Our approach to compute the next time-step size for components C_0 to C_4 from Figure 5. Imagine that we perform an initial step estimation (1, the minimum step size is denoted by a dotted line). Consider further that we use the maximum possible step size of C_2 to be the next step size for all components (2). Although this seems to be feasible at first sight, we have to take all data dependencies into account. This reveals that we cannot interpolate an intermediate value for C_0, since it depends on buffer information from the previous iteration. Therefore, the maximum step size is computed using all components that are directly reachable via back edges (3).

Note that we must always have access to the source item values at the beginning of the time step in order to interpolate between the source and target values. If the application does not use double buffering by default, we have to copy all relevant values that have to be considered during interpolation into separate global-memory buffers.

4.1 Leveraging Shared-Memory Caches

Depending on the computational load induced by an interpolation function, it can be advantageous to cache already interpolated values in shared memory. This frees up resources in terms of required bandwidth and ALUs and makes

them available to the underlying component implementation. This can reduce the overhead of our method in the case of expensive interpolations (see Section 5). Without explicit modeling of such a cache, the GPU will have to recompute the interpolation function for each item access (see Figure 7). Furthermore, this will trigger two additional loads from global memory for each item access. In this scope, L1 and L2 caches help to reduce the actual number of global-memory transfers automatically on modern GPUs [26] (see also Section 5).

Figure 8 visualizes several access patterns of an imaginary component in the scope of a single thread group. In our implementation, each thread just caches its associated item value(s) that will be accessed in the scope of the component in shared memory (1-to-1 thread-item mapping). Components in our scope are already programmed while having GPU-like architectures in mind. Therefore, they typically perform coherent memory accesses that are very close in global memory with respect to the current thread index (local access window). These cases can be covered by our shared-memory cache without the need for sophisticated program transformations. Unfortunately, we have to check that a certain access to a particular item is included in our cache (via an *if-branch*), which imposes additional runtime overhead. More advanced static program analyses can help to determine the actual access pattern(s) in order to realize more efficient caches in the future. This can even help to avoid on-the-fly cache-boundary checks that can be expensive with respect to thread divergences and register usage.

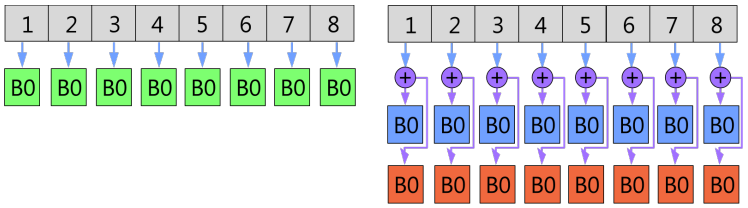

Fig. 7. An imaginary thread group of 8 threads accesses items from a buffer $B0$ in global memory (left). Using an interpolation function triggers two loads from global memory (from the source and target buffers respectively) for each access (right). Furthermore, the loaded values need to be interpolated to get the actual item value at a specific point in time.

4.2 Hiding Different Access Patterns using Views

From a practical point-of-view, every component can be modeled as a specific object-oriented *class* that implements a particular interface. An abstract pseudo-code interface definition we use for our components is shown in Listing 1.1 to get a better understanding about the general functions a component needs to provide. We propose this generic interface definition that distinguishes between *component-data* and *item-data* views. This allows us to clearly separate component-specific (uniform for each component type C) and item-specific (varies from item to item) information. Since the functions *ComputeNumSteps* and *Evaluation* work on abstract data views, it is easily possible to replace an

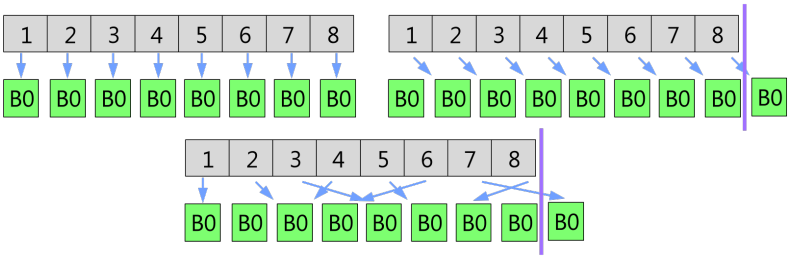

Fig. 8. Different access patterns for an imaginary thread group of 8 threads. Coherent memory accesses with respect to each thread while accessing a shared-memory cache (top). Random memory accesses that are not coherent (bottom). The vertical bars indicate the boundaries of the shared-memory cache. Accesses that cross these boundaries will be cache misses and have to be resolved via expensive global-memory loads and interpolation-function applications.

item access with an interpolated value (that is even cached in shared memory) without touching the component implementation. Another advantage of having separate views is the improvement of static-program-analysis results with respect to potential aliases. This is due to the fact that each view has a very limited scope that is it only accessible within a single function. We do not allow storing views of any kind within member fields of a component to reduce the risk of bugs and to limit potential aliases in practice.

Listing 1.1. An abstract component interface definition used in our implementations in pseudo C# code. Generics in C# (or templates in C++) allow us to hide the actual data-view implementation from each component realization.

```
interface IComponent<TComponentImplementation>
  where TComponentImplementation :
    IComponent<TComponentImplementation>
{
  // Initializes internal fields by loading data from global memory.
  static TComponentImplementation Init(int index, ComponentView source);

  // Computes the number of steps that this component can perform.
  // Required information is loaded from the provided source data view.
  int ComputeNumSteps<TDataView>(TDataView source);

  // Evaluates this component by applying the given number of simulation
      steps.
  // Computes results are written to the provided target data view.
  void Evaluate<TDataView>(TDataView target, int numSteps);
}
```

4.3 Algorithm

The main kernel that wraps the actual functionality of a component C is shown in Algorithm 2. It is designed to be specialized by a compiler (via meta-programming techniques or code generation) to generate an individually instantiated GPU program. The first lines allocate all required shared-memory resources based on knowledge from the dependency graph. Afterwards, we perform a padded grid-stride loop (to avoid thread divergences) over the whole range of C. The body of the outer loop is another loop that performs the required small steps in the scope

of a larger Δt time step. Important to mention is the group-wide computation of the next step size that will be applied to all threads in the group. This is required since all threads can access our shared-memory caches and need to have access to consistent interpolated information. If the presented shared-memory caches are not suitable for the application scenario, several parts of the algorithm are not required (like the lines 1–3 and 10). This even affects the computation of the upcoming local step-size consisting of several reductions and group barriers. These operations will be no longer required in such a case. Further optimizations (like loop unrolling, multiple components per kernel or even thread compaction) can be applied based on the actual component implementation to increase occupancy and to improve runtime performance.

4.4 Implementation Details

We have implemented our algorithm in C# using the ILGPU[2] compiler for all GPU kernels. Regarding code generation, we used a custom pre-compilation step to gather all data dependencies between all components. Afterwards, we generate all required C# kernels for each component and the whole adaptive-time-stepping driver code to execute all components in a specialized simulation loop. Each component will be wrapped in a particularly specialized time-stepping algorithm based on Algorithm 2. This includes the generation and allocation of shared-memory caches and inlining of all interpolation functions. The adaptive time-step sizes for each component are realized with the help of specialized kernels using efficient warp reductions and atomic operations [25, 26].

5 Evaluation

The evaluation section covers two different application scenarios inspired by particle-based simulations. Each scenario is described with the help of a component dependency graph to give detailed insights into the modeled simulation structure. In order to avoid hard-to-reproduce benchmarks, we use component implementations that are based on matrix-matrix multiplications to generate computational load per item. Moreover, memory-accesses to neighboring particles (that are often accessed in SPH-based simulations [5]) always consider 9 neighboring items. We used two different interpolation functions (linear and cubic-spline) to emulate less- and more-expensive interpolation computations. Similarly, we used a varying number of items $|R|$ (the range, particles in these scenarios) to analyze the scaling behavior of our adaptive time-stepping approach. Most important for the evaluation are the number of steps we can adaptively perform. In order to be close to application scenarios, we vary the number of steps continuously (computed using a uniform random distribution) for all components from the intervals $[1, \dots 3]$. We have not included larger intervals to show realistic performance measurements that do not assume optimistic properties of the underlying simulation. This emulates common scenarios in which we can sometimes perform larger steps, while other situations require small step sizes to satisfy all domain-specific simulation constraints.

[2] www.ilgpu.net

Algorithm 2: Our adaptive time-stepping algorithm for component C

```
   Input: maxNumSteps, source, originalSource, target
 1 nextStepSize := shared memory int[1];
   /* Shared memory allocations for all intermediate values          */
 2 sharedMemoryCache0 = shared memory Type0[group size];
   /* ...                                                            */
   /* Perform a grid-stride loop over the padded value range of C    */
 3 for i := global index; i < padded_range(C); i += grid size * group size
    do
       /* Optional: synchronize group members to improve the memory
          access pattern on some GPU architectures                    */
       /* group barrier                                               */
 4     localSource, localTarget := source, target;
 5     numPerformedIterations := 0;
 6     for stepIdx := 0; stepIdx < maxNumSteps; do
 7         view := new CachedDataView<Interpolation Function>(
 8             localSource, localTarget, originalSource, i,
 9             maxNumSteps / (stepIdx + 1) as float,
               /* References to shared memory allocations             */
10             sharedMemoryCache0, ...);
           /* Wait for all cached values to be available and initialize
              the next maximum step size                              */
11         if is first thread of group then
12             nextStepSize := maxNumSteps - stepIdx;
13         end
14         group barrier;
           /* Check component precondition for the current value      */
15         instanceStepSize := max(int);
16         C c := ⊥;
           /* Compute next common step size for all group threads     */
17         if i < range(C) then
18             c := C.Init(i);
19             instanceStepSize := c.ComputeNumSteps(view);
20             warpWideStepSize := warp reduce min(instanceStepSize);
21             if is first lane of warp then
22                 atomic min nextStepSize, warpWideStepSize;
23             end
24         end
25         group barrier;
26         stepSize := nextStepSize;
           /* Apply component with the next common step size          */
27         if i < range(C) then
28             c.Evaluate(view, stepSize);
29         end
           /* Advance step index and wait for all threads             */
30         stepIdx += stepSize;
31         numPerformedIterations++;
32         Swap localSource, localTarget;
33         group barrier;
34     end
35     if numPerformedIterations is even then
36         Perform a parallel group-wide copy operation of affected information
            from source to target buffers;
37     end
38 end
```

We measured four different algorithms: *Simple* (fixed step size of 1), *Trad. Adaptive* (adaptive time stepping based on prior work, see Section 3), *HIP-NoCache* (our method without shared-memory caches) and *HIP* (our method with caches enabled). Every setup has been evaluated using two GPUs from NVIDIA for all benchmarks (a GeForce GTX 980 Ti and a GeForce GTX 1080 Ti). Furthermore, every performance measurement is the median execution time of 100 simulation runs, each performing a maximum number 100 simulation steps (when using a time-step size of 1). All execution times are measured in milliseconds (ms).

5.1 Gravity-like Simulation

This evaluation scenario covers a gravity-like simulation that leverages three components $C0, C1$ and $C2$ (see Figure 9). The first component conceptually computes particle-specific information, whereas the second one iterates over neighboring particles in memory and prepares accumulated results for C_2. The last component accesses neighboring information from B_0 and B_1 and computes item updates.

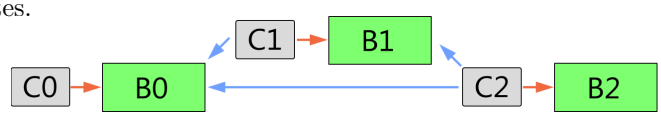

Fig. 9. Component dependency-graph of the first evaluation scenario.

Table 1 shows the performance measurements for the discussed configurations. In the presence of a small range $|R|$, the traditional method cannot improve performance significantly. It can only improve performance of about 4.5% to 7% in the case of 64k items. Using our approach allows us to increase the performance approx. 12% to 17% in comparison to the traditional adaption approach. Adding shared-memory caches does not hurt in most cases and is able to improve the runtime by approximately 2% on average using cubic-spline interpolation. However, modern GPU architectures do not seem to benefit from the explicit caching method in the case of simple linear interpolation functions.

Table 1. Performance measurements of the first evaluation scenario.

$\|R\|$	I	**Algorithm**	**980 Ti**	σ	**1080 Ti**	σ
16384	-	**Simple**	**136.36**	18.21	**83.45**	9.24
*	-	**Trad. Adaptive**	**135.49**	4.14	**82.67**	1.16
*	Linear	**HIP-NoCache**	**118.02**	5.44	**72.32**	3.62
*	Spline2	*	**121.63**	5.14	**74.02**	3.21
*	Linear	**HIP**	**116.48**	4.22	**72.16**	2.91
*	Spline2	*	**117.25**	6.60	**72.97**	2.53
65536	-	**Simple**	**399.47**	13.62	**245.88**	0.68
*	-	**Trad. Adaptive**	**381.34**	23.13	**229.36**	0.51
*	Linear	**HIP-NoCache**	**340.04**	1.74	**203.84**	0.12
*	Spline2	*	**342.30**	3.87	**209.74**	0.03
*	Linear	**HIP**	**332.88**	1.65	**204.35**	0.09
*	Spline2	*	**335.36**	4.16	**205.74**	0.12

5.2 PBD-like Simulation

This evaluation scenario is inspired by a PBD-like simulation that involves seven different components. C_0 computes basic information per particle. This can be seen as a more expensive position-prediction step derived from PBD. As before, C_1 prepares accumulated neighborhood information that is used for an imaginary collision-detection step in C_2. C_3 and C_4 use SPH-based calculations across all neighboring particles to simulate a reasonable workload per item. Afterwards, C_5 iterates over all neighbors while accessing items in B_4 and B_3. C_6 computes final item updates using B_5.

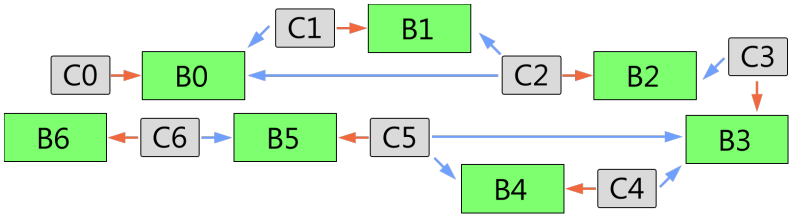

Fig. 10. Component dependency-graph of the second evaluation scenario.

Table 2 depicts the performance numbers for the discussed evaluation configurations. In all cases, the traditional adaption approach performs slower than the non-adapted version. This is due to the fact that the probability is lower than in the previous scenario to execute a larger time step (see also Section 3) as we are using more components. However, we are able to improve the performance approx. 16% to 22% in comparison to the non-adapted version. Again, shared-memory caches can help to improve the runtime (in this scope of up to 6%) on older GPU architectures.

Table 2. Performance measurements of the second evaluation scenario.

$\|R\|$	I	**Algorithm**	**980 Ti**	σ	**1080 Ti**	σ
16384	-	**Simple**	**327.50**	82.33	**200.95**	11.43
*	-	**Trad. Adaptive**	**346.83**	11.32	**212.75**	6.59
*	Linear	**HIP-NoCache**	**282.70**	26.43	**171.10**	3.78
*	Spline	*	**293.63**	16.05	**175.12**	13.57
*	Linear	**HIP**	**282.88**	19.57	**172.27**	10.72
*	Spline	*	**284.50**	12.92	**178.05**	18.46
65536	-	**Simple**	**1006.99**	85.99	**602.42**	2.49
*	-	**Trad. Adaptive**	**1016.65**	0.23	**615.63**	8.23
*	Linear	**HIP-NoCache**	**826.18**	16.77	**503.14**	2.94
*	Spline	*	**878.80**	18.51	**516.66**	0.50
*	Linear	**HIP**	**825.83**	18.62	**504.14**	0.19
*	Spline	*	**827.86**	19.63	**510.84**	3.24

6 Conclusion

We present a new approach to realize generic and domain-independent time-step adaptive GPU-based simulations. Our concept is based on several components that define the actual simulation loop. A static program analysis resolves data dependencies between read/write accesses of all components. The determined

dependency graph is then used to generate specialized kernels that access interpolated values at intermediate time steps. In this scope, our generic component model allows to hide the actual memory-access implementation logic. This enables us to integrate our shared-memory-based caching concept and domain-specific interpolation functions touching the component implementations.

Our approach scales perfectly with the complexity of the underlying simulation model. Even in non-optimal use cases consisting of a few components, our approach was able to outperform fixed step sizes and the traditionally used conservative step-adaption method. However, our integrated caching concept is only beneficial if the simulation uses computationally expensive interpolation functions. This should not be necessary on modern GPUs when using linear or cubic-spline value-estimation methods. In case of more sophisticated simulations, we were able to significantly improve performance in comparison to prior work of up to 22% on non-optimistic benchmarks for our idea. If we assume more optimistic scenarios, we will be able to achieve impressive performance speedups in particular on large-scale simulations involving many different components. This makes our method a perfect extension to every modern GPU-based simulation that wants to benefit from adaptive time steps.

Probably the primary downside of our approach is the fact that we might introduce simulation errors with respect to adaptively adjusted time steps. As in all related papers, the actual time-step size calibration itself is tightly coupled to the domain and is usually adjusted by a domain expert. Therefore, we argue that this is not a disadvantage that has been introduced by our method. Furthermore, we have the ability to express domain-specific criteria to limit the step size.

In the future, we would like to extend the concept to support more advanced program analyses. This would allow us to simplify shared-memory-cache accesses considerably. In addition, we want to experiment with different caching concepts that even works on the warp-level in register space and in shared memory.

Acknowledgments

The authors would like to thank T. Schmeyer, A. Bosch and J. Bayer for their feedback on the paper, even in the scope of very challenging and stressful times.

References

1. Adams, B., Pauly, M., Keiser, R., Guibas, L.J.: Adaptively sampled particle fluids. ACM Transactions on Graphics (2007)
2. Carey, V., Estep, D., Johansson, A., Larson, M., Tavener, S.: Blockwise Adaptivity for Time Dependent Problems Based on Coarse Scale Adjoint Solutions. Siam Journal on Scientific Computing (2010)
3. Gander, M., Halpern, L.: Techniques for locally adaptive time stepping developed over the last two decades. Lecture Notes in Computational Science and Engineering (2013)
4. Garcia, V.M., Liberos, A., Climent, A.M., Vidal, A., Millet, J., González, A.: An adaptive step size GPU ODE solver for simulating the electric cardiac activity. In: 2011 Computing in Cardiology (2011)
5. Groß, J., Köster, M., Krüger, A.: Fast and Efficient Nearest Neighbor Search for Particle Simulations. In: Eurographics Proceedings (2019)

6. Hong, W., House, D.H., Keyser, J.: An Adaptive Sampling Approach to Incompressible Particle-Based Fluid. In: Theory and Practice of Computer Graphics (2009)
7. Horvath, C.J., Solenthaler, B.: Mass Preserving Multi-Scale SPH. Pixar Technical Memo 13-04, Pixar Animation Studios (2013)
8. Ihmsen, M., Akinci, N., Gissler, M., Teschner, M.: Boundary Handling and Adaptive Time-stepping for PCISPH (2010)
9. Kay, D., Gresho, P., Griffiths, D., Silvester, D.: Adaptive Time-Stepping for Incompressible Flow Part II: Navier–Stokes Equations. SIAM Journal on Scientific Computing (2010)
10. Kelager, M.: Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics (2006)
11. Köster, M., Groß, J., Krüger, A.: Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction. International Journal of Parallel Programming (06 2020)
12. Köster, M., Groß, J., Krüger, A.: FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs. In: 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019) (2019)
13. Köster, M., Groß, J., Krüger, A.: Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs. In: Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019) (2019)
14. Köster, M., Krüger, A.: Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications. International Journal of Computer Graphics & Animation (2016)
15. Köster, M., Leißa, R., Hack, S., Membarth, R., Slusallek, P.: Code Refinement of Stencil Codes. Parallel Processing Letters (PPL) (2014)
16. Köster, M., Schmitz, M., Gehring, S.: Gravity Games - A Framework for Interactive Space Physics on Media Facades. In: Proceedings of the International Symposium on Pervasive Displays (2015)
17. Köster, Marcel and Krüger, Antonio: Screen Space Particle Selection. In: Eurographics Proceedings (2018)
18. Macklin, M., Müller, M.: Position Based Fluids. ACM Trans. Graph. (2013)
19. Macklin, M., Müller, M., Chentanez, N., Kim, T.Y.: Unified Particle Physics for Real-time Applications. ACM Trans. Graph. (2014)
20. Mayr, M., Wall, W., Gee, M.: Adaptive time stepping for fluid-structure interaction solvers. Finite Elements in Analysis and Design (2018)
21. Monaghan, J.J.: Smoothed particle hydrodynamics. In: Annual Review of Astronomy and Astrophysics (1992)
22. Müller, M., Charypar, D., Gross, M.: Particle-based Fluid Simulation for Interactive Applications. In: Symposium on Computer Animation (2003)
23. Müller, M., Heidelberger, B., Hennix, M., Ratcliff, J.: Position Based Dynamics. J. Vis. Comun. Image Represent. (2007)
24. Müller, M., Solenthaler, B., Keiser, R., Gross, M.H.: Particle-Based Fluid-Fluid Interaction. In: Symposium on Computer Animation (2005)
25. NVIDIA: Faster Parallel Reductions on Kepler (2014)
26. NVIDIA: CUDA C Programming Guide v10 (2019)
27. Pounders, J., Boffie, J.: Analysis Of An Adaptive Time Step Scheme For the Transient Diffusion Equation (2015)
28. Solenthaler, B., Gross, M.: Two-scale Particle Simulation. In: ACM Siggraph (2011)
29. Solenthaler, B., Pajarola, R.: Predictive-Corrective Incompressible SPH. In: ACM Siggraph (2009)
30. Zhang, Y., Solenthaler, B., Pajarola, R.: Adaptive Sampling and Rendering of Fluids on the GPU. In: Eurographics Proceedings (2008)