1. INTRODUCTION

Graphics processing units (GPUs) have seen a rapid development in performance in recent years, now boasting Tera-range peak floating point operations counts. For example, the NVIDIA GTX 480 GPU provides 1345 GFlop/s of computing, compared to the fastest multi-core CPU, e.g., the Intel Core i7 965 XE, that performs at about 70 GFlop/s [13]. Initially, GPUs were used primarily by the gaming community. The appearance of Compute Unified Device Architecture (CUDA) with a new parallel programming model and instruction set architecture, leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on CPUs [18].

Nowadays, GPUs are being used as computational resources for various scientific problems, e.g., gravitational direct N-body simulations [2], solving Kepler's equation [5], and the solution of stochastic differential equations [10]. Michalakes et.al. [14] were the first to apply GPUs in Numerical Weather Prediction (NWP). They adapted the physics module of the Weather Research and Forecast (WRF) model [24], which originally is in Fortran, to run on a NVIDIA 8800 GTX GPU using CUDA.

In this paper, we investigate another computationally intensive module in NWP, namely the dynamics routine of the HIRLAM weather forecast model [9]. It codes the grid point, explicit Eulerian primitive equations [17]. In contrast to the physics module mentioned above, in the dynamics the calculations on a certain grid point require information from horizontally neighboring grid points. The conventional parallel implementation of the dynamics using distributed memory parallel architectures results in communications between the processors due to dependencies between variables that reside in different processor memories [9]. On GPUs in CUDA the grid points are organized in threads and blocks, which means that information should be exchange between threads and blocks. In general this also means that synchronization between threads and blocks should be necessary. Between threads this is no problem, but between blocks this is impossible in CUDA, since the blocks have to be executable independently, in an arbitrary and undefined order. To solve this issue we investigate two options to build independent blocks. The first is by recalculation: when a thread in a block needs information from a thread in a neighboring block, the information is recalculated by the thread itself. This results in multiple calculations of the same intermediate result(s). We will call this method "recalculation". The second option is to split the dynamics routine into several kernels, where each kernel consists of blocks that can execute independently. After the calculation of information that is needed by neighboring blocks, the kernel is stopped and a the next one that uses this information is started. The synchronization between blocks is now established by the fact that a kernel completes before the next one starts. In this way all required information is first calculated before it is needed by some thread or block. We call this method "multi-kernel". It results in an additional overhead of kernel invocations. The difference between the recalculation method and the multi-kernel method resembles that between macro- and micro-parallelism, respectively.

We will investigate if and how the overhead, incurred in these two methods, will affect the performance of an implementation of the dynamics routine on GPUs. Can we achieve a significant speedup on GPUs for the dynamics even though there are dependencies between blocks, dependencies that do not exist for the physics studied by Michalakes et.al. [14]?

In general, the performance of a CUDA program depends on the way the data is distributed over the threads, and how threads are organized into blocks. In our CUDA version of the dynamics we can achieve this by specifying statically the number of grid points per thread, and the number of threads per block as parameters before compiling. The optimal code is found by tuning the values for these parameters empirically.

Transferring data between CPU and GPU is costly. The multi-kernel approach allows the use of multiple CUDA streams, to overlap execution of one kernel with transferring input for a subsequent kernel, or output from a previous one. We investigate this method in much detail.

In Section 2 we summarize the related work. Section 3 covers the background of the HIRLAM weather forecast model and GPU-based computing. Section 4 describes how to convert from Fortran to C and then to CUDA code. The experiments are shown in Section 5. In Section 6 we present how

CTADEL can generate the optimal CUDA streams program to overlap execution and data transfer. Conclusions are given in Section 7.

2. RELATED WORK

The use of GPUs as a low-cost, low-power, and very high efficient computational resource for high performance computing has received great attention in recent years. In the area of weather and climate prediction, several papers have dealt with the exploitation of GPUs. Michalakes et.al. [14] were among the first when they adapted the physics module of the Weather Research and Forecast (WRF) model [24], originally written in Fortran, to run on an NVIDIA 8800 GTX GPU using CUDA. This change brings a speedup of 10 over an Opteron 2.4 GHz CPU for the physics module itself, and speeds up the whole weather forecast model by a factor of 1.23. After that, several studies on using GPUs for weather and climate forecasting have emerged. In [22, 23], Shimokawabe et.al. implemented the dynamics routine of the Japan Meteorological Agency (JMA) weather forecast model [11] on a Tesla S1070. The governing equations for the dynamics in JMA are the compressible non-hydrostatic equations written in flux form. Comparing to the original code on an Opteron 2.4 GHz CPU, the implementation on the GPU is 80 times faster. Govett et.al. [7, 8] converted the dynamics routine of the NOAA Weather Model [16] from Fortran to CUDA using a code translator tool. They obtained a speedup of 24 on a GTX 280 GPU over an Intel Harpertown 2.8 GHz CPU. Kelly [12] ported the Community Atmosphere Model at the National Center for Atmospheric Research [15] to GPUs. The Community Atmosphere Model is a global atmospheric model used for climate and weather research. It is also the primary atmospheric component of the Community Climate System Model, which simulates the planet's atmospheric and entire climate system, including land surface, ocean, and sea ice. Benchmarked on a NVIDIA 9800 GX2 GPU, a 14 times performance improvement was achieved compared to the original implementation on an Intel Core 2 2.6 GHz.

In [26], we have implemented the dynamics routine of the HIRLAM weather forecast model on a NVIDIA 9800 GX2 GPU. We converted the original Fortran to C and CUDA by hand, straightforwardly, without much concern about optimization. In this paper we investigate several optimizations for the CUDA implementation.

Most elapsed time is actually spent on transferring data over the PCI bus between CPU and GPU. The above papers did not propose any approach to reduce the influence of these transfer times. In this paper, we surmount this limitation by using so-called CUDA streams.

3. BACKGROUND

3.1. The HIRLAM weather forecast model

HIRLAM [9] stands for HIgh Resolution Limited Area Model and is a state-of-the-art numerical weather analysis, forecasting and post-processing system. It is a cooperative project of Denmark, Estonia, Finland, Iceland, Ireland, the Netherlands, Norway, Spain, and Sweden. The aim of this project is to develop and maintain a numerical short-range weather forecast system for operational use by the participating institutes. The HIRLAM forecast system is now used in routine weather forecasting by the national weather centers of Denmark, Finland, Ireland, The Netherlands, Norway, Spain and Sweden.

The HIRLAM weather forecast model is one of the three components in the HIRLAM system. It consists of two main modules, the *dynamics* and the *physics*. The dynamics routine solves the so-called primitive equations, describing conservation of horizontal momentum, energy and mass (air and water), and the ideal gas law. The solution method to solve this set of equations numerically is based on an Eulerian explicit method using grid points [17]. The physics scheme parameterizes the effects of sub-grid-scale processes like turbulence, and of the non-adiabatic processes like radiation, phase transitions and exchange with the earth surface. As opposed to the physics scheme, the dynamics equations contain horizontal gradients. Computationally, this implies that the physics routines do not require horizontal communications, but the dynamics does.

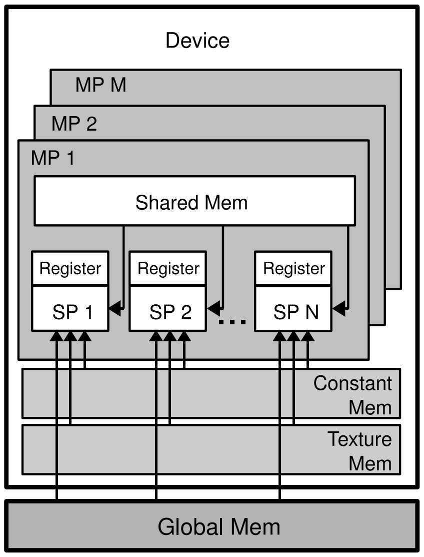

Figure 1. GPU Architecture

3.2. GPU-based computer architecture

This subsection describes some details about the hard- and software models of GPUs in a CUDA context.

Hardware model An NVIDIA GPU consists of a group of so-called *multiprocessors* (MP), where each MP contains of a number of SIMD ALUs. A single ALU is called a *scalar processor* (SP). The structure of a GPU is presented in Figure 1.

A GPU is equipped with a multi-level memory. Each SP has access to local *registers*. Each MP has a local *shared* memory. Access to shared memory is as fast as to registers, and takes 2-4 clock cycles. A *global* memory is available for all SPs of a GPU. Access to global memory takes hundreds of cycles. In addition, a *texture* and *constant* memory are available for every MP. Both texture and constant memory are read-only.

Software model Programming NVIDIA GPUs for general-purpose applications is enabled by CUDA. In the CUDA context, a CPU is called *host*, and a GPU a *device*. A CUDA application

Table I. Mapping between memory hierarchy and CUDA programming model

memory	location	scope	lifetime
register	on chip	thread	thread
local	off chip	thread	thread
shared	on chip	all threads in block	block
constant	off chip	all threads + host	allocated from host
texture	off chip	all threads + host	allocated from host
global	off chip	all threads + host	allocated from host

runs in two parts. The code that executes on the device is called the *kernel*. The other part works on the CPU as *host* code, providing control over data transfers between CPU and GPU, the invocation of kernels, and other parts that do not run on the GPU.

A number of *threads* execute a kernel in single program multiple data (SPMD) mode. A group of threads are batched into a *block*. Within a block threads are organized in a 1-, 2-, or 3-dimensional structure. All threads in a block can share data through the shared memory and synchronize their execution by a barrier construct. Several blocks are organized into a 1- or 2-dimensional *grid*. An important restriction is that CUDA does not allow to synchronize between blocks. A grid is launched on the device by scheduling its blocks on MPs. Each MP can execute one or more blocks. Within a block threads are grouped in *warps* consisting of 32 threads executing in SIMD fashion.

The memory model of the CUDA programming model is presented in Table I. Threads have their own register file and local memory. Local memory is only used if the register file is full. However access to local memory is slow, since it is off chip. Threads in a block share the same shared memory. All threads in a grid have access to constant, texture, and global memory.

4. CODE CONVERSION

The dynamics routine DYN of the HIRLAM weather forecast model, generated by the code generator CTADEL, consists of 800 lines of Fortran 77 code. In this section we present the manual conversion of this routine from Fortran to C and then to a CUDA program.

4.1. CUDA code translation

CUDA is an extension of C, therefore we chose to rewrite the Fortran program to C before creating CUDA. The translation from Fortran to C is performed straightforwardly by obvious grammar conversions.

The translation from C to CUDA consists of creating host code and kernel(s). The host code includes allocation and deallocation of memory on the CPU and GPU. It transfers the input for the DYN routine from the CPU to the GPU before the kernel is started, and the output back after kernel completion. And it defines the thread structure and invokes the kernel. In CUDA a set of primitives supports these tasks.

Figure 2 illustrates the host code of the CUDA program. In this figure, *Tg* and *T_tg* represent the variables on the GPU for respectively *T* and *T_t* which reside on the CPU. Allocation and deallocation are specified by “cudaMalloc” and “cudaFree”, respectively, and the transfer of data between CPU and GPU is specified by “cudaMemcpy”. The threads within a block are arranged three-dimensionally, and the blocks within a grid two-dimensionally, by the structures “dimBlock” and “dimGrid”.

The kernel DYN invoked in the host code is illustrated in Figure 3. The kernel is very similar to C code, with the “__global__” declaration specification meaning that this function resides on the device and is only callable from the host.

Finally, the Fortran, and hence the C program, uses two- and three-dimensional arrays, whereas the CUDA language allows only single dimensional array addressing. As an example, in Figure 3 the three-dimensional array holding the temperature *T*[i][j][k] is converted to the one-dimensional *T*[Index3(i,j,k)], where Index3 is defined as

```
//Allocate memory (T_t, T) on CPU:
  float *T_t, *T;
  T_t = (float *)malloc(size-of-T_t);
  T   = (float *)malloc(size-of-T);
//Allocate memory (T_tg, Tg) on GPU:
  float *T_tg, *Tg;
  cudaMalloc ((void**)&T_tg, size-of-T_tg);
  cudaMalloc ((void**)&Tg,   size-of-Tg);
//Transfer input from CPU (T) to GPU (Tg):
  cudaMemcpy(Tg, T, size-of-Tg, cudaMemcpyHostToDevice);
//Define structure of threads:
  dim3 dimBlock(blocksize_x,blocksize_y,blocksize_z);
  dim3 dimGrid(gridsize_x,gridsize_y);
//Invoke kernel
  DYN<<<dimGrid,dimBlock>>>(Tg,T_tg,...);
//Transfer output from GPU (T_tg) to CPU (T_t):
  cudaMemcpy(T_t, T_tg, size-of-T_tg, cudaMemcpyDeviceToHost);
//Deallocate memory (T_tg, Tg) on GPU:
  cudaFree (T_tg);cudaFree (Tg);
//Deallocate memory (T_tg, Tg) on GPU:
  free(T_t);free (T);
```

Figure 2. Host code

```
  Index3(i,j,k) = i + MLON * (j + k * MLAT)
```

with MLON and MLAT the sizes of the first and second dimension of the original three-dimensional array, respectively. We note that C and CUDA have a row-wise memory layout. Therefore, in order to have consecutive memory access, in the Index3 definition, we use i as the most inner loop.

To reduce the cost of index calculations, we have performed several optimizations on the Index3 function, such as predefine Index3(i,j,k) as a local value, then calculate the index of the next grid point by adding an appropriate stride. However, these optimizations do not improve the performance much, which indicates that the CUDA compiler is aware of this kind of optimizations.

```
//Define kernel DYN:
 __global__
 void DYN(float *Tg,float *T_tg,...)
  {
   for (k=lk;k<=uk;k++){
    for (j=lj;j<=uj;j++){
     for (i=li;i<=ui;i++){
      T_tg[Index3(i,j,k)]= Tg[Index3(i,j,k)]...;
     }
    }
   }
  }
```

Figure 3. Kernel code

Synchronization between blocks In the DYN routine the calculations on a certain grid point require information from horizontally neighboring grid points. For example, to calculate the temperature $T\_t[i,j,k]$ we need pressure $p[i,j,k]$, $p[i+1,j,k]$, and $p[i,j+1,k]$. In a straightforward thread structure, where each thread processes one grid point, $T\_t[i,j,k]$ and $p[i,j,k]$ would be calculated by thread (i,j,k). This implies that thread (i,j,k) cannot start before threads $(i+1,j,k)$ and $(i,j+1,k)$ have completed their calculation of $p[i+1,j,k]$ and $p[i,j+1,k]$, respectively. The required synchronization of threads can be achieved by using two kernels to calculate p and T_t separately. The synchronous execution of kernels ensures the synchronization of threads. If we take all synchronizations into account, we have to split the DYN routine into 14 kernels. We call this the multi-kernel approach.

Alternatively, within blocks synchronization and communication between threads are possible and cheap, by using barrier $\_\_syncthreads()$ and local memory, respectively. However, threads on block boundaries cannot synchronize with their neighbor threads in neighboring blocks, because in CUDA blocks may be executed independently in any order [18]. To avoid data dependencies between blocks, we can build an independent block by letting thread (i,j,k) calculate $p[i+1,j,k]$ and $p[i,j+1,k]$ itself instead of waiting for them to be calculated by threads $(i+1,j,k)$ and

$(i, j+1, k)$, respectively. Because $p[i+1, j, k]$ and $p[i, j+1, k]$ are also evaluated by threads $(i+1, j, k)$ and $(i, j+1, k)$, there is a risk of race conflicts in memory access. We avoid that by treating them as local variables. We call this is the recalculation approach. The simplest implementation of the recalculation method is to let each thread calculate three local variables $p00 = p[i, j, k]$, $p10 = p[i+1, j, k]$ and $p01 = p[i, j+1, k]$, at the expense of having to calculate p three times at every grid point. In Section 5 we will compare this fairly simple code to the multi-kernel approach; but in the future we intend to investigate the far more complicated code in which the recalculation is only done by the threads on a block boundary.

Number of grid points per thread and number of threads per block The performance of a CUDA program depends on the distribution of data, in this case grid points, over threads and threads over blocks. The best choice for these options depends on the application. We enable empirical optimizations by introducing flexibility on these distributions into our CUDA program.

Memory usage From the multi-level memory hierarchy (see Table I) we use shared memory only for loop boundaries. All input and output of DYN are kept in global memory. In the future, we will investigate optimizations by using shared, texture, and constant memory.

4.2. CUDA streams

In Figure 2, the host code includes the three main steps: transfer all inputs from CPU to GPU; launch the kernel DYN; and transfer all outputs from GPU to CPU. These operations are invoked serially. We call this the non-stream CUDA code.

With multiple CUDA streams [21] we can overlap calculations (run a kernel) with transferring data between CPU and GPU (asynchronous copy). There are obvious requirements for a CUDA streams program: the inputs have to be present on the GPU before they can be referred to and the outputs can only be transferred after they have been calculated. Moreover, if a calculation of a stream refers to inputs/values which are transferred/calculated by a different stream, then synchronization between the streams is required.

Table II. The queued order of the two-stream CUDA code, where overlapping of calculation with data transfer is denoted by the shading.

Code line	**Stream 1**	**Stream 2**
1	Transfer(*A*)	
2	Transfer(*B*)	
3	Transfer(*ps*)	
4	Calculation(*p* (*A*,*B*,*ps*))	Transfer(*hxv*)
5	Transfer(*u*)	
6	Calculation(*t60* (*u*,*p*))	Transfer(*hyu*)
7	Transfer(*v*)	
8	Calculation($t64$ (*v*,*p*))	Transfer(*hxt*)
9		Transfer(*hyt*)
10	Synchronization	
11		Calculation(*ps_t*)
12		Transfer(*ps_t*)

Table II is an example of a two-stream strategy, where we calculate ps_t using inputs (*A*,*B*,*ps*,*u*,*v*,*hxv*,*hyu*,*hxt*,*hyt*). Pressure p is a user-defined variable, $t60$ and $t64$ are temporary variables, introduced by common subexpression elimination. We built this strategy on two ideas. Firstly, the queued order of operations is arranged to obtain a maximum overlap between calculation and data transfer. Secondly, the transfer of input, output and calculation of a variable should preferably be in the same stream to avoid synchronization. This strategy needs only one synchronization, at code line 10.

The calculation of ps_t, the surface pressure tendency, is the first step in DYN. The full DYN also does full three-dimensional tendencies of temperature, wind, and humidity. In the full DYN, while

ps_t is calculated in stream 2, stream 1 is used to transfer input for the next step in DYN, and the "Transfer(ps_t)" is postponed until the first execute-step which is not concurrent to a input transfer.

To profit from streaming, the memory on the host has to be page-locked [18], implying that it will not be swapped out of memory. We do not have to fear performance loss due to this page-locking [19], because on the one hand we only need 200 MB page-locked memory out of 12 GB available memory, and on the other hand the CPU is not heavily loaded since all calculations are executed on the GPU.

4.3. The CUDA implementation for multiple GPUs

The 1536 MB memory of the GTX 480 GPU limits the domain size of DYN to around 15 million grid points. To run larger domains, we implement the DYN routine on multiple GPUs.

Multiple GPUs can be configured to be bound to one or more CPUs. Because of data dependencies, when using several CPUs those CPUs must communicate with each other. The data communication between processors has been presented in detail in [25], including the following steps: determine the sizes of the data areas to be communicated (the halo zones); copy the required communicated data into the buffers; exchange with the neighboring processors; and fill the received data in the halo zones. Here, we limit ourselves to a configuration of multiple GPUs, all bound to the same CPU. In this configuration, we define a set of threads on the CPU, one for each GPU. To avoid confusion: we use the term "thread on the CPU" for a thread that executes on the CPU, to distinguish with the CUDA thread that executes on the GPU.

The parallelization over the threads on the CPU is carried out by openMP [20]. Figure 5 presents the structure of the CUDA code for multiple GPUs.

The parallelization over the GPUs consists of domain decomposition, kernel execution, and data communication between the GPUs. The kernel is similar to that for a single GPU. The domain is decomposed simply as in Figure 4: given g GPUs, the problem domain is divided in the x-direction into g sub-domains of approximately equal size. The calculations on a certain GPU require information from the neighboring sub-domains. Because the GPUs cannot communicate directly

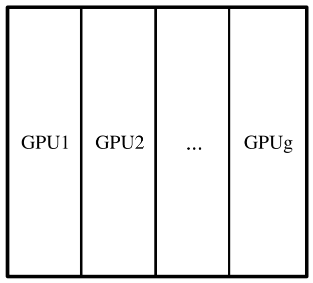

Figure 4. Domain decomposition over g GPUs

```
// Get the number of available GPUs
   cudaGetDeviceCount(&number_of_gpus);
// Create as many CPU threads as number of GPUs
   omp_set_num_threads(number_of_gpus);
// Calculate in parallel on multiple GPUs
   #pragma omp parallel
    {
     Decompose domain & distribute data to GPUs
     Execute CUDA kernel on multiple GPUs
     Communicate data between GPUs
    }
```

Figure 5. Structure of CUDA Code for Multiple GPUs

with each other, the data exchange is via the CPU: for example, to communicate data from GPU1 to GPU2 the data are first transferred from GPU1 to the CPU, and then from the CPU to GPU2. In the sequel, we will use the term "transfer" for data transfer from CPU to GPU and back, and "communication" for data transfer between GPUs. Currently "communication" has to be by data "transfers", in the formal sense of these words, but the important distinction is by the fact that the CPU does nothing with the "communicated" data, except passing them on from one to another GPU.

Table III. Specifications of the NVIDIA GTX 480 GPU

Scalar Processors (SP)	480
Multiple Processors (MP)	15
Processor clock (MHz)	1401
Global memory (MB)	1536
Shared memory per MP (KB)	64
Memory clock (MHz)	1848
Memory bandwidth (GB/sec)	177.4
Peak single-precision performance (Gflops)	1345
Maximum number of threads per block Recommended maximum number of threads per block	1024 512
Number of threads that can coordinate global memory accesses	32

5. EXPERIMENTS

The experiments are conducted on a platform with two Intel Core Quad i7-940 CPUs at 2.93 GHz, 12 GB of RAM, and a single NVIDIA GeForce GTX 480 GPU card using driver version 195.36.15, CUDA Toolkit version 3.0. The operating system is CentOS Linux 5.0 with the 2.6.18 UNIX x86_64 kernel. As compiler we use *gcc* version 4.1.2 for C and *nvcc* version 3.0 for CUDA. The specifications of GTX 480 GPU are listed in Table III.

The base code for this investigation is the DYN routine of the HIRLAM weather forecast model. In this paper we will compare the performance of the CUDA code with the C program. As metrics for the performance of the different codes we use the following times: the execution time is the time to execute DYN on the GPU; the transfer time is the time to transfer inputs/outputs between CPU and GPU; the communication time is the time to exchange data between the GPUs; and the elapsed time of the CUDA program which includes the execution time, the transfer time, and in case of multiple GPUs the communication time.

To verify the correctness of the C and all CUDA implementations we compare the calculated results in all experiments with those calculated by the original Fortran code. These comparisons show that the C and CUDA codes reproduce the output of the Fortran program within meteorological accuracy, and the different versions of CUDA codes reproduce bit-wise identical output.

5.1. The CUDA code on a single GPU

Multi-kernel or recalculation approach

As mentioned in Section 4, to avoid data dependency between blocks, we can build independent blocks by recalculation or split the DYN routine into multiple kernels. In order to assess the overhead costs of the two methods, we compare the execution times of these approaches with a program in which we neither split the DYN routine nor recalculate variables. We call this program the single kernel. This code does not have additional costs of recalculation or multi-kernel invocation. However, because blocks cannot synchronize, the program only produces correct output if we assign the whole computational domain to a single block. Hence, we run the three programs on a single block grid with the recommended maximum number of 512 threads (32×16), where each thread processes a column of 64 grid points. This results in domain of $32 \times 16 \times 64$ grid points.

Table IV shows that the execution time for this domain increases from 1.86 ms to 1.97 ms by going from single to multi-kernel. The overhead for recalculation is 4 times bigger.

Nowadays, domains for operational weather forecasting have dimensions in the order of $1000 \times 1000 \times 60$. Domains of this size do not fit into the memory of a single GPU. In [26], we used the $500 \times 200 \times 64$ domain as basic configuration. Timings for this domain are also included in Table IV. Note that the single kernel program, due to the lack of synchronization between blocks, produces wrong results. It must not be used for any other purpose than for quantification of the overhead needed for synchronization. The overhead of the multi-kernel approach is hardly bigger for this domain than for the small one, but recalculation is much more expensive. This is explained by the fact that the overhead of recalculation is more or less proportional to the number of grid

Table IV. The execution times (in ms) of the multi-kernel and recalculation approaches. For reference we also show times for the single kernel program. However, it does not produce correct results on the bigger domains (indicated by *); see text.

Code	$32 \times 16 \times 64$	$500 \times 200 \times 64$	$512 \times 192 \times 64$
Single kernel	1.86	52.21*	43.81*
Multiple kernel	1.97	52.34	43.95
Recalculation	2.31	98.07	84.72

points, because each grid point requires the same number of recalculations, whereas in the multi-kernel approach the overhead is more or less proportional to the number of kernel invocations, which does not depend on the domain size.

To fully utilize all threads in a block and all blocks in a grid, the domain sizes in all directions should be multiples of the dimensions of a block. Global memory is most efficiently accessed if full warps, each of 32 threads, access it consecutively [18]. Hence we tried a domain of $512 \times 192 \times 64$. Table IV shows that indeed a domain with these “nice” dimensions is processed much more efficiently, and the synchronization overheads remain roughly the same. Hence, we will use this domain in the sequel, and also, of course, we will use the multi-kernel approach.

Varying the number of grid points per thread and the number of threads per block

Let the subdomain treated within one thread be called the thread domain and GX, GY, and GZ denote the number of grid points in the x-, y-, and z-direction of the thread domain, respectively. In this subsection we will look for optimal values of GX, GY, and GZ.

To preserve the advantage of the full warp consecutive memory access (see Table III and [18]), GX should be equal to 1. Therefore, in the next paragraphs we vary GY and GZ, while keeping GX=1.

Table V lists the execution times of the CUDA code for various values of GY, while keeping GZ fixed at 64. We use 64 threads per block. The big jump in performance when going from GY=1 to

Table V. Execution times (in ms) as function of the number GY of grid points in the y-direction per thread, with 64 threads per block

GY	**1**	2	4	8	16
$512 \times 192 \times 64$	**43.9**	53.5	55.4	57.9	59.9

Table VI. Execution times (in ms) as function of the number GZ of grid points in the z-direction per thread, with 64 threads per block

GZ	Block	Time
64	64x1x1	33.9
32	32x1x2	**33.3**
16	16x1x4	43.7
8	8x1x8	72.3
4	4x1x16	134
2	2x1x32	307
1	1x1x64	566

GY=2 is explained by the introduction of an additional loop in the code. The modest increase in execution time when GY is further increased is probably the result of an increased register and local memory usage.

Table VI shows the execution times with different values of GZ. Based on the previous experiment, we fix GY=1 and again use 64 threads per block. Because a grid is only two-dimensional, the block has to cover the vertical size of the computational domain, which is 64 grid points. The third dimension of the block is therefore 64/GZ. Because the number of threads per block is also 64, the first dimension of the block then turns out to be equal to GZ. The results in the table, with an optimum at GZ=32, reflect clearly that the first dimension of the block should be a multiple of the warp size.

The first and second dimensions of a block can be chosen arbitrarily provided that the total number of threads per block does not exceed 1024 [18]. In Table VII, we vary the number of threads per

Table VII. Execution times (in ms) as function of the number GZ of grid points in the z-direction per thread for different block structures

GZ	Block	Time	Block	Time	Block	Time	Block	Time
64 threads block								
64	16x4x1	42.9	32x2x1	34.9	64x1x1	33.9		
32	16x2x2	41.7	32x1x2	**33.3**				
16	16x1x4	43.7						
128 threads block								
64	16x8x1	45.9	32x4x1	35.4	64x2x1	33.8	128x1x1	33.3
32	16x4x2	45.3	32x2x2	34.1	64x1x2	**33.2**		
16	16x2x4	43.6	32x1x4	33.8				
8	16x1x8	47.5						
256 threads block								
64	32x8x1	34.7	64x4x1	34.5	128x2x1	31.7	256x1x1	31.6
32	32x4x2	35.2	64x2x2	32.5	128x1x2	**31.5**		
16	32x2x4	35.2	64x1x4	33.5				
8	32x1x8	38.4						
512 threads block								
64	64x8x1	35.5	128x4x1	31.9	256x2x1	31.2	512x1x1	31.2
32	64x4x2	36.1	128x2x2	31.5	256x1x2	**31.1**		
16	64x2x4	35.1	128x1x4	33.1				
8	64x1x8	41.3						

block. The results show that larger blocks are more efficient. Additionally, a second block dimension equal to 1 and third equal to 2 seems to be optimal. In summary, we conclude that $1 \times 1 \times 32$ is the best choice for the thread domain and $256 \times 1 \times 2$ is for the block structure. In the remainder of this paper, we will use these structures for the experiments.

We note that in the above experiments, the block may be two- or three-dimensional. But in the DYN routine, the surface pressure tendency is two-dimensional and hence its kernel may be

Table VIII. Elapsed times (in ms) on a GTX 480 GPU, as function of domain size, and, for reference, that of the C code on the Intel i7-940 CPU. Thread domain: 1x1x32. Block structure: 256x1x2.

Domain		$512 \times 192 \times 64$	$512 \times 288 \times 64$	$512 \times 384 \times 64$	$512 \times 480 \times 64$
C (-O2)		2610	3950	5280	6620
CUDA	Transfer	42.2	60.7	81.4	99.8
nonstream	Execution	31.1	46.9	62.7	78.1
	Total	73.3	107.6	144.1	177.9
CUDA streams		46.9	70.1	94.4	118.1

executed by a two-dimension block, although other variables are calculated by a three-dimension block. However, the advantage of this optimization and complication is small (around 1%).

CUDA streams

We have discussed how to overlap kernel execution with data transfer using CUDA streams in Subsection 4.2. Table VIII shows the elapsed time of the non-stream and CUDA streams program on a NVIDIA GTX 480 GPU, and of the C code on an Intel i7-940 CPU for different domain sizes. We fix the domain size in the x- and z-direction at 512 and 64 grid points, respectively, and vary the number of grid points in the y-direction.

With the non-stream method, we measure an elapsed time of 73.3 ms for the domain of $512 \times 192 \times 64$ grid points, consisting of 31.1 ms execution time and 42.2 ms for data transfer from/to the GPU. The elapsed time of the CUDA streams program is 46.9 ms. Hence, by overlapping, the total time is reduced by 26.4 ms. On the elapsed time this is a modest reduction (36%), but it should be realized that overlapping cannot reduce the elapsed time below the maximum execution time and transfer time, which is 42.2 ms in this case. The result may also be read as: the overhead of data transfer is reduced from 42.2 ms to 15.8 ms, a reduction of 63%. Furthermore, it is likely that without increasing the elapsed time, the number of calculations may be increased, by which a more accurate model may become feasible, but now without the usual extra computer costs.

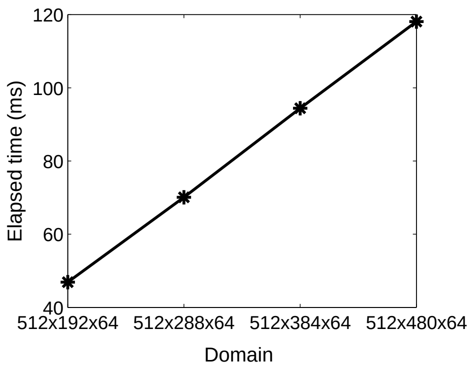

Figure 6. Elapsed time of the CUDA streams program as a function of the domain size

Another observation from Table VIII is that we obtain a speedup of 55 for the CUDA streams program over the C code. This speedup is much larger than the difference between the peak performance of the GTX 480 GPU and Intel i7-940 CPU, which is around 20 times. One of the reasons is that we use *gcc* version 4.1.2 which does not support openmp. This gcc version is only able to exploit one of the 4 cores of the Intel i7-940 CPU. With a *gcc* version that implements openmp, the performance of the C code may be better. However, the goal of this investigation is to assess the performance of an implementation of a weather forecast model on GPUs. Hence, we do not want to spend time on adapting the C code to fully utilize the Intel i7-940 CPU. Even if we use 4 cores, the performance of the C code will increase at most 4 times, and the performance gain of the CUDA program over that C code is still more than a factor of 10.

Figure 6 demonstrates that the elapsed time of the CUDA streams program scales extremely well with the domain size.

5.2. The CUDA code for multiple GPUs

For multiple GPUs runs, we implemented the DYN routine on 1, 2, and 4 GPUs controlled by one CPU host. As a result of the domain decomposition, each GPU performs calculations on a sub-domain equivalent to the ratio between the whole computational domain and the number of GPUs.

The experiments for multiple GPUs code are performed on a machine with two Intel Xeon E5620 CPUs at 2.40 GHz, 16 GB of RAM, and four NVIDIA GeForce GTX 480 GPU cards using driver

Table IX. Execution times (in ms) of the non-stream CUDA code on multiple GPUs

Domain	1 GPU	2 GPUs	4 GPUs
$512 \times 192 \times 64$	31.57	15.75	7.76
$512 \times 288 \times 64$	47.08	23.40	11.60
$512 \times 384 \times 64$	63.54	31.20	15.19
$512 \times 480 \times 64$	78.53	38.76	18.76

version 260.19.26, CUDA Toolkit version 3.2. The operating system is Linux 2.6.31-22-generic. As compiler we use gcc version 4.4.1 for C and nvcc version 3.2 for CUDA.

Table X. Transfer and communication times (in ms) of the non-stream CUDA code on multiple GPUs

Domain		1 GPU	2 GPUs	4 GPUs
$512 \times 192 \times 64$	Communication		0.41	0.43
	Transfer Input	18.31	9.19	8.95
	Transfer Output	24.52	12.24	10.37
$512 \times 288 \times 64$	Communication		0.56	0.57
	Transfer Input	27.42	13.65	13.57
	Transfer Output	36.78	18.34	18.90
$512 \times 384 \times 64$	Communication		0.67	0.69
	Transfer Input	36.43	18.27	17.78
	Transfer Output	48.91	24.48	23.38
$512 \times 480 \times 64$	Communication		0.78	0.78
	Transfer Input	45.49	22.96	22.83
	Transfer Output	61.08	30.54	25.32

Table IX shows the execution times of the non-stream CUDA code. We see that the execution times decrease by a factor equal to the number of GPUs. This is because each GPU has to process a subdomain of which the size decreases proportional to the number of GPUs.

Table X shows the transfer and communication times. We observe small communication times. This is due to the fact that the amount of communicated data is small (0.1 MB), because we only exchange data in the halo zone, which is one or two grid points wide [3]. Furthermore, because of our choice to decompose the whole computational domain only in the x-direction (see Figure 4), we only need to communicate with the neighboring GPU in the x-direction. The amounts of communicated data, therefore, are the same in case of 2 or 4 GPUs. As a result, the communication times of 2 and 4 GPUs runs are similar. The size of input/output data depends on the GPU subdomain which decreases proportional to the number of GPUs. Therefore, we expected that the transfer times reduce by a factor equal to the number of GPUs. This is only true for the case of 2 GPUs runs, where the transfer times decrease by a factor 2. The transfer times of 4 GPUs runs are lightly smaller that those of 2 GPUs runs.

With 4 GPUs, we obtain a speed-up of 2.7 over a single GPU. This is a poor efficiency of parallelization. The main efficiency loss is seen in the data transfers from CPU to GPU and from GPU to CPU. Options to improve this efficiency are: 1) Apply the CUDA streams program techniques in a multiple GPU configuration, by which data transfers will be overlapped with calculations. 2) Bind one GPU to one CPU. So data is transferred from one CPU to one GPU or from one GPU to one CPU. The disadvantage of this approach is that it will result in additional communications between the CPUs.

6. AUTOMATIC GENERATION OF THE OPTIMAL CUDA STREAMS PROGRAM

In Subsection 5.1, we have shown that a particular choice of a CUDA streams program to overlap the kernel calculations with the data transfers increases the performance up to 36%. That CUDA streams program was created by hand which is difficult to optimize and complicated to maintain. Therefore, in this section we present our extension to the code generation tool CTADEL [4] to automatically generates the optimal CUDA streams program. Firstly, we show a technique that generates CUDA streams program for a general problem. Then we apply it for the dynamics routine of the HIRLAM weather forecast model.

6.1. Generating the optimal CUDA streams program for the general problem

Suppose that we need to execute a number of kernels on a GPU. Each kernel produces a number of output results from a number of input values. Because the CPU and the GPU cannot access memory of each other, we need to transfer input values from the CPU to the GPU before the calculation and output results from the GPU to the CPU after the calculation. To reduce the data transfer time, we only transfer inputs/outputs of the program while keeping intermediate values on the GPU. The calculation in a kernel may require an output value of another kernel as its input. Therefore the execution of kernels has to be arranged in an appropriate order, given by the *dependency graph*. This dependency graph is a directed acyclic graph (DAG) that represents the dependencies between the calculation of kernels and their inputs/outputs. In the dependency graph, an invocation of a kernel is called a *node*. A node takes a set of input values and/or output data of other nodes and uses them to create one or more output values. Figure 7 is an example of a dependency graph. In this figure a node is denoted by an oval, an input/output value is denoted by a rectangle, and an arrow represents a dependence of a node on its input. In this example, the calculation of kernel A needs data from inputs 1 and 2, the calculation of kernel B needs data from inputs 2, 4 and 5, and the calculation of kernel C needs data from output of kernels A and B and data from inputs 3 and 4. The output value of the total process consists of outputs 1 and 2 of kernels A and C, respectively.

Based on the dependency graph, we can generate the optimal CUDA streams program through four steps as follows:

1. *Generate the calculation streams*. A kernel has to be invoked after another kernel if its calculation uses output of that kernel. Therefore, from the dependency graph we can determine the invocation order of kernels. We group kernels into a list that reflects the invocation order of kernels: the first kernel in the list is invoked firstly, then the second kernel, and so on. We

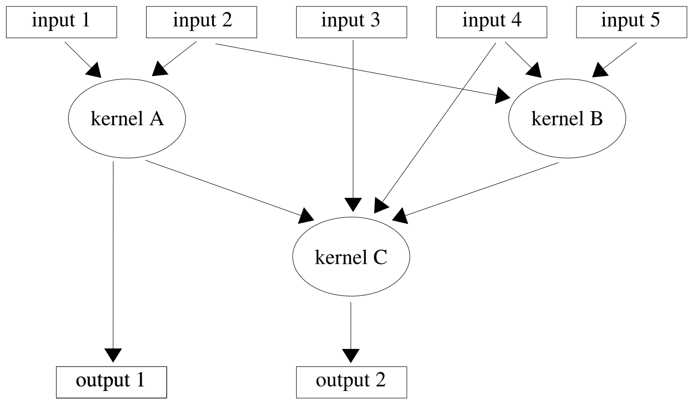

Figure 7. An example data dependency graph

call this list of kernels the *calculation stream*. In the example in Figure 7, the calculation of kernel A does not depend on output of any other kernel. Hence, kernel A can be invoked firstly. Similarly, we can also invoke kernel B firstly. The calculation of kernel C needs the outputs of kernels A and B. Therefore, kernel C has to be invoked after kernels A and B. As a result, we can create two calculation streams: [kernel A,kernel B,kernel C] and [kernel B,kernel A,kernel C].

2. *Generate the CUDA streams codes*. The CUDA streams code is generated by adding the input/output transfers required by each kernel to the calculation stream. To have overlap, we split the transfer of inputs/outputs and the calculation of kernels into two CUDA streams so that one stream executes a kernel while the other stream is transferring data. To avoid confusion, we note that a CUDA streams code is the host code of a CUDA streams program which consists of invocation of kernel executions and data transfers. In term of a host code, these invocations are processed serially. But in term of a CUDA streams code, these invocations are organized into two CUDA streams which are executed in parallel. To describe the generation of the CUDA streams code, we will use the terms “previous kernel” and “next kernel” for the kernel that stands in front of and behind the current kernel, respectively, and the term “later kernel” for one of the kernels that stand behind the current kernel in the calculation stream. The CUDA streams code is created based on four principles. First, the inputs have to

be present on the GPU before they can be referred to. Therefore, we add the input transfers of the next kernel to overlap with the calculation of the current kernel. Second, the outputs can only be transferred back after their actual calculation. Hence, we add the output transfers of the current kernel to overlap with the calculation of the next kernel. Third, if the calculation of a kernel in a CUDA stream refers to inputs/values which are transferred/calculated in a different CUDA stream, a synchronization is needed before the invocation of this kernel. Similarly, a synchronization is needed before a transfer of an output that is calculated in a different CUDA stream. And fourth, to have maximum overlap, we add more data transfers to overlap with an expensive kernel and remove transfers if the kernel execution time is smaller than the transfer time(s).

The approach to generate the CUDA streams code is as follows:

- Transfer the inputs of the first kernel in the calculation stream;
- For all kernels:
  - Synchronize if necessary;
  - Transfer the inputs of the next kernel in a CUDA stream;
  - While the input transfers time is smaller than the kernel execution time, transfer one input of the later kernel, or one calculated output if all inputs have been transferred;
  - Execute the kernel in another CUDA stream;
- Transfer the outputs of the last kernel.

Table XI shows an example CUDA streams code that is generated from the calculation stream [kernel A,kernel B,kernel C] of the example dependency graph in Figure 7. Firstly, inputs 1 and 2 of kernel A are transferred in CUDA stream 1. Next, the calculation of kernel A is overlapped with the transfers of input 4 and 5 of kernel B, the calculation of kernel B is overlapped with the transfers of input 3 of kernel C and output 1 of kernel A. Kernel B invoked in CUDA stream 2 uses input 2 which is transferred in CUDA stream 1, hence a synchronization is included before the invocation of kernel B. Similarly, since kernel C invoked in CUDA stream 1 uses output of kernel B, which is invoked in CUDA stream 2, a

Table XI. Example CUDA streams code generated from the calculation stream [kernel A,kernel B,kernel C]. Overlap of the kernel calculation and input/output transfers is indicated by the kernel calculation and input/output transfers in the same row.

Code line	**CUDA stream 1**	**CUDA stream 2**
1	transfer input 1 transfer input 2	
2	kernel A	transfer input 4 transfer input 5
3	synchronization	
4	transfer input 3 transfer output 1	kernel B
5		synchronization
6	kernel C	
7	transfer output 2	

synchronization is needed before the invocation of kernel C. Finally, output 2 of kernel C is transferred.

3. *Derive the theoretical time of the CUDA streams codes*. The theoretical time of a CUDA streams code is an aggregation of the transfers and execution time in case of non overlap. The kernel execution time and the input/output transfer time can be extrapolated by the number of operations executed by each kernel and the size of data to be transferred, respectively, or can be found empirically. Examples are the theoretical time of row 1 in Table XI which is equal to the total transfer time of inputs 1 and 2, or the theoretical time of row 6 which is the execution time of kernel C. If there is overlap between a kernel execution and data transfers, then the theoretical time is equal to the maximum value of those two times. For example the theoretical time of row 2 in Table XI is the maximum value of the kernel A execution time and the total transfer time of inputs 4 and 5.

4. *Generate the optimal CUDA streams program*. In general from a dependency graph we can generate many calculation streams, corresponding with many generated CUDA streams codes with different theoretical times. As a final step, we choose the CUDA streams program that has the minimal theoretical time and generate the corresponding optimal CUDA streams program.

6.2. *Generating the optimal CUDA streams program for the DYN routine*

In this section we present how to generate the optimal CUDA streams program for the dynamics routine (DYN) of the HIRLAM weather forecast model.

The DYN routine

The DYN routine of the HIRLAM weather forecast model solves the so-called primitive equations that describe the conservation of horizontal momentum, energy and mass (air and water), and the ideal gas law, on the grid points. Specifically, the DYN routine calculates the tendency of surface pressure ps_t, humidity q_t, temperature T_t, and wind components (u_t,v_t) from the inputs including vertical hybrid coordinate (A,B), coriolis effect f, metric coefficient (hxu,hxv,hxt,hyu,hyv,hyt), surface geopotential $phis$, surface pressure ps, humidity q, temperature T, and wind (u,v). The dependency graph of the DYN routine is shown in Figure 8. In this graph, p, lnp, $etap$, E, Z, and phi are the user-defined variables; $t25$, $t60$, and $t64$ are temporary variables introduced by common subexpression elimination.

Table XII. Execution time of the kernels (microsecond) for the domain of $512 \times 192 \times 64$ grid points

Kernel	p	$etap$	lnp	E	Z	phi	ps_t	$t60$	$t64$	$t25$	q_t	T_t	u_t	v_t
Time	230	1044	438	786	1626	1044	54	359	666	791	4641	10920	4295	4290

To derive the theoretical time of the CUDA streams code, we need information about the time to execute the kernels and the transfer time for the input/output data. The kernel execution times and the input/output transfer times depend on the computational domain. We measure these times for the domain of $512 \times 192 \times 64$ grid points. The results are shown in Tables XII and XIII.

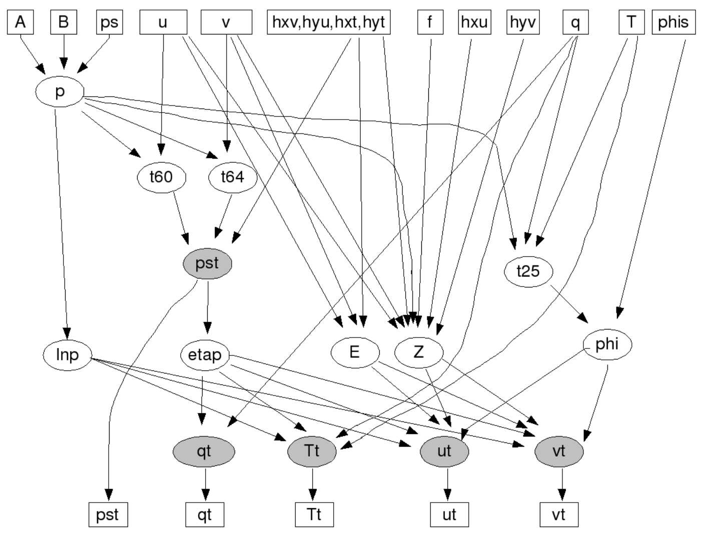

Figure 8. Dependency graph of the DYN routine. The inputs and outputs are denoted by a rectangular. The calculated variables are denoted by a oval. An arrow shows the dependence of a kernel calculation on an input or a calculated value of other kernel or the dependence of an output on a calculated value of a kernel.

Table XIII. Transfer time of the input/output (in microsecond) for the domain of $512 \times 192 \times 64$ grid points

Variable	Size	Transfer time
1D inputs (A, B)	256 B	18
2D inputs $(ps, f, phis, hxu, hxv, hxt, hyu, hyv, hxt)$	384 KB	85
3D inputs (q, T, u, v)	24 MB	4400
2D outputs (ps_t)	384 KB	85
3D outputs (q_t, T_t, u_t, v_t)	24 MB	4130

Table XIII shows that for the 3D data, transferring the input from the CPU to the GPU is more expensive than the output from the GPU to the CPU. This result is also observed in other studies such as [1, 6].

Generating the optimal CUDA streams program

The optimal CUDA streams program is generated by the following steps:

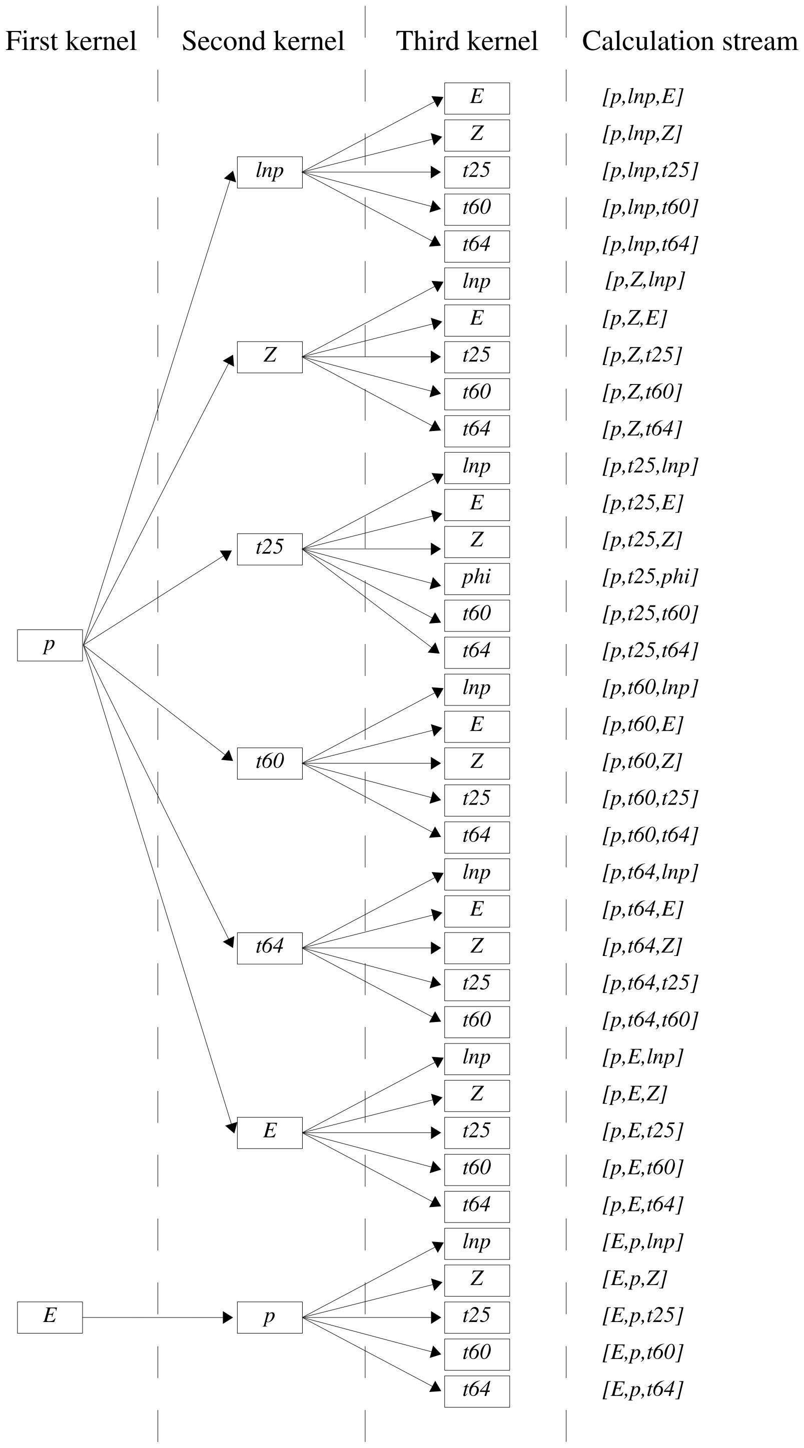

Figure 9. The calculation streams generated for the first three kernels. An arrow shows which kernel can be invoked after the calculation of another kernel.

Table XIV. The optimal CUDA streams code. Overlap of the kernel calculation and input/output transfers is indicated by the kernel calculation and input/output transfers in the same row.

CUDA stream 1	**CUDA stream 2**	**Time (μs)**
Transfer(A,B,ps)		121
Calculation(p)	Transfer(hxv) Transfer(hyv)	230
Synchronization		
Transfer(v)	Calculation(lnp)	4400
Calculation($t64$)	Transfer(u)	4400
Transfer(hxu) Transfer(hyu) Transfer(hxt) Transfer(hyt)	Calculation($t60$)	359
	Synchronization	
Calculation(ps_t)		54
Synchronization		
Transfer(f) Transfer($phis$)	Calculation($etap$)	1044
Calculation(Z)	Transfer(q)	4400
Transfer(T)	Calculation(q_t)	4641
	Synchronization	
Calculation(T_t)	Transfer(q_t)	10920
Synchronization		
Transfer(ps_t)	Calculation(E)	791
Calculation($t25$)		786
Synchronization		
	Calculation(phi)	1044
	Synchronization	
Calculation(u_t)	Transfer(T_t)	4295
Transfer(u_t)	Calculation(v_t)	4290
	Transfer(v_t)	4130
Total theoretical time		45905

1. Based on the dependency graph, CTADEL generates the calculation streams. Figure 9 shows, as an example, how the first three kernels of the calculation streams are generated. As first kernel p or E can be chosen, because the calculations of them do not use outputs of any other kernel. Next, if we choose p as first kernel, the kernels lnp, Z, $t25$, $t60$, $t64$, and E can

Table XV. The theoretical and run time (in ms) of the generated CUDA streams program, and, for reference, the run time of the handwritten program, on a GTX 480 GPU.

	Theoretical time	Run time
Handwritten CUDA streams program		46.9
Generated CUDA streams program	45.91	46.03

be selected as second kernel, because the calculations of these kernels only depend on the calculation of p. Similarly, if we choose lnp as second kernel, the kernels E, Z, $t25$, $t60$, and $t64$ can be chosen as third kernel.

2. From the calculation streams, CTADEL generates the CUDA streams codes by adding the input/output transfers as described in Subsection 6.1.
3. Next, CTADEL derives the theoretical time of all generated CUDA streams codes.
4. Finally, CTADEL chooses the CUDA streams code with the minimal theoretical time to generate the optimal CUDA streams program.

Table XIV shows the flow in the CUDA streams program which has the minimal theoretical time of 45905 μs.

Results

To assess the performance of the generated CUDA streams program we run this program on the system that we used in Section 5 and compare it with the performance of the handwritten CUDA streams program.

The domain and threads structure are chosen the same as in Section 5: the computational domain consists of 512x192x64 grid points; the thread domain contains 1x1x32 grid points, and the block has 256x1x32 threads. The elapsed times are shown in Table XV.

Because we cannot measure the synchronization time, we did not include it in deriving the theoretical time of the CUDA streams code. As a result, the real run time of the generated program is larger than the theoretical time as we can see from Table XV. However, this small difference between the real run time and the theoretical time indicates that the synchronization time is small and can be eliminated in deriving the theoretical time of the CUDA streams code.

The optimal CUDA streams program generated by CTADEL is slightly faster than the handwritten program. This demonstrates that, although we have done many optimizations, the handwritten program is not optimal. It also shows that CTADEL can generate efficient CUDA programs that are even better or at least comparable in performance with optimized handwritten programs.

7. CONCLUSIONS

We have demonstrated that the execution on a GPU yields an order of magnitude performance improvement for the dynamics of HIRLAM weather forecast model.

We manually translated the original Fortran code to C and from there to CUDA. Compared to the C code on a single core of an Intel i7-940 CPU, the CUDA program executes 55 times faster on a NVIDIA GTX 480 GPU. The performance improvement is expressed in elapsed time, but, given the low price and the power consumption of a GPU card, also price/performance and power consumption per performance are impressive.

The largest domain that fits on the memory of one GTX 480 GPU is 512x480x64 grid points, which is a factor 4 smaller than current operational domains. Therefore, we implemented the CUDA program on 4 GPUs, all bound to the same CPU, and obtained a speed-up of 2.7. This result is disappointing, but by applying the CUDA streams program techniques or binding more CPUs it is expected that the speed-up can be improved significantly.

The experiments have shown that the use of CUDA streams reduces the elapsed time by 36% as a result of overlapping kernel execution and data transfer. The overlapping reduces the overhead of data transfer by 63%, and probably will allow future model extensions without increasing the elapsed time. The handwritten CUDA streams program, however, is difficult to optimize and complicated to maintain. Hence, we extended CTADEL to generate the optimal CUDA streams program. The experimental results showed that the generated program is slightly more efficient than the handwritten program.

REFERENCES

1. V. Allada, T. Benjegerdes, and B. Bode, Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster, Cluster Computing and Workshops, CLUSTER '09, pp 1-9, 2009.
2. R.G. Bellemana, J. Bédorf, and S. F. Portegies Zwart, High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, *New Astronomy 13*, pp 103-112, 2008.
3. R.A. van Engelen, L. Wolters, and G. Cats, Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling, *IEEE Journal of Computational Science and Engineering*, Vol. 4, No. 3, pp 22-31, 1997.
4. R.A. van Engelen, Ctadel: A Generator of Efficient Numerical Codes, *PhD thesis*, Universiteit Leiden, The Netherlands, 1998.
5. E.B. Ford, Parallel algorithm for solving Kepler's equation on Graphics Processing Units: Application to analysis of Doppler exoplanet searches, *New Astronomy 14*, pp 406-412, 2009.
6. M. Fatica, Accelerating Linpack with CUDA on heterogeneous clusters, ACM ICPS 383, pp 46-51, Stockholm, Sweden, 2009.
7. M. Govett, C. Tierney, J. Middlecoff, and T. Henderson, Using Graphical Processing Units (GPUs) for Next Generation Weather and Climate Models, *Climate and Atmospheric Sciences (CAS) workshop*, September 2009, Annecy, France.
8. M. Govett, Using GPUs for Weather and Climate Models, *Earth System Prediction Capability (ESPC) Workshop*, September 2010, New Orleans, USA.
9. HIRLAM, http://hirlam.org/.
10. M. Januszewskia and M. Kostur, Accelerating numerical solution of Stochastic Differential Equations with CUDA, *Computer Physics Communications*, Vol. 181, pp 183-188, 2009.
11. Japan Meteorological Agency, http://www.jma.go.jp/jma/indexe.html.
12. R. Kelly, GPU Computing for Atmospheric Modeling, *Computing in Science and Engineering*, Vol. 12, No. 4, pp. 26-33, July/August, 2010.
13. L. Lugmayr, Intel Core i7 965 Extreme Edition Review, http://www.i4u.com/article21670.html.
14. J. Michalakes and M. Vachharajani, GPU Acceleration of Numerical Weather Prediction, *Parallel Processing Letters*, Vol. 18, No. 4, pp 531-548, 2008.
15. National Center for Atmospheric Research, http://ncar.ucar.edu/
16. National Weather Service, NCEP Central Operations, http://www.nco.ncep.noaa.gov/
17. F. Nebeker, Calculating the Weather: Meteorology in the 20th Century, *Academic Press*, ISBN 0-12-515175-6, vii+255p, 1995.
18. NVIDIA, CUDA Programming Guide, V.3.1, 2010.
19. NVIDIA, NVIDIA CUDA C Best Practices Guide, CUDA Toolkit 2.3, July 2009.
20. http://openmp.org/wp/.
21. J. Sanders and E. Kandrot, CUDA by Example, *Addison-Wesley*, ISBN 0-13-138768-5, xix+290p, 2010.
22. T. Shimokawabe, GPU Acceleration of Weather Prediction Model, *Computing with GPUs, Cells, and Multicores Workshop*, May 2010, Tokyo, Japan.
23. T. Shimokawabe, T. Aoki, and J. Ishida, GPU Acceleration of Meso-scale Atmosphere model ASUCA, *In Proceeding of the 9th World Congress on Computational Mechanics and 4th Asian Pacific Congress on Computational Mechanics (WCCM/APCOM 2010)*, Sydney, Australia, July 2010.
24. W.C. Skamarock, J.B. Klemp, J. Dudhia, D.O. Gill, D.M. Barker, W. Wang, and J.G. Powers, A description of the advanced research WRF version 2, *Technical Report NCAR/TN-468+STR*, National Center for Atmospheric Research, January 2007.
25. Van Thieu Vu, Gerard Cats, and Lex Wolters, *Overlapping communications with calculations*, In Proceedings of the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2009), CSREA Press, pp 487-493, Las Vegas, USA, July, 2009.
26. V.T. Vu, G. Cats, and L. Wolters, GPU Acceleration of the Dynamics Routine in the HIRLAM Weather Forecast Model, *In Proceedings of the 2010 International Conference on High Performance Computing & Simulation (HPCS 2010)*, Caen, France, July 2010.