The handle [http://hdl.handle.net/1887/18622](http://hdl.handle.net/1887/18622) holds various files of this Leiden University dissertation.

**Author:** Vu, Van Thieu  
**Title:** Opportunities for performance optimization of applications through code generation  
**Issue Date:** 2012-03-22
Chapter 7

An overview of research results achieved

Numerical Weather Prediction (NWP) is computationally intensive; increasing demands on quality and details of weather forecast and climate simulations require continued work on the deployment of the most recent hardware and software. To avoid total code redesign with every new hard- or software development we have chosen the approach of code generation.

The use of code generation has several advantages. Firstly, it takes less time to implement a new problem. Hence, the researcher is free to concentrate on solving the problem instead of spending time in writing code. Secondly, it is easy to maintain. If there are requirements to change the problem, the user only needs to modify the specification in the code generator instead of having to rewrite the code. Furthermore, a code generator can generate efficient code for different computer architectures. If the hardware cost model of the target computer architecture is included, the code generator will generate code optimized for that computer architecture. However, code generation has disadvantages as well. The biggest disadvantage is the need for the scientist to learn a new way of thinking and to learn a new language. The second is that each code generator is usually targeted to a specific application domain, with a specific programming model and/or language. Moreover, the time to generate code for a serious problem may become substantial.

Code generation with CTADEL has been demonstrated to be very powerful for finite difference methods and for vector- and parallel hardware. In this thesis we investigated extensions of CTADEL to other integration schemes and to new developments on the hardware side. In Chapter 2 we used CTADEL to generate code for a finite element method. In Chapters 5 and 6 we chose GPUs as a demonstration bench to investigate how CTADEL can be applied to adopt existing software to new hardware, without the need of a total redesign.
of the software. Also, in Chapters 3 and 4 we performed research into efficiency improvements that were hitherto out of the question because of the complicated code they require. However, the code complications are not prohibitive if those complications are automatically generated rather than programmed manually. In our investigations we used NWP software for benchmarking. Most of the work was on HIRLAM dynamics, except in Chapters 2 and 4 where CTADEL was used to generate code for the Shallow-Water equations.

Below we summarize the topics that have been investigated and the results that we have obtained during our research. We started the research by demonstrating that CTADEL can generate code for different integration schemes. Till now, CTADEL has been applied to generate efficient code for several numerical schemes in the weather forecast model. These codes use finite difference methods for discretization. Finite difference methods are restricted to handle rectangular shapes and simple alterations, since in these methods the derivatives at a grid point are approximated from the function values at other grid points. If the geometry of the grid points is irregular, it is difficult to obtain a high-order accuracy approximation with finite difference methods. On the other hand, finite element methods are formulated using variational principles, which involve minimizing a continuous function defined over the area or volume of the element. The function is described by the element’s nodal values, and the integration required to minimize the function is done numerically. Therefore, using finite element methods in case of complex geometries and irregular physical structures is more simple. In Chapter 2 we investigated the extension to CTADEL to generate code for Galerkin finite element methods. Then we applied to generate code for the Shallow-Water equations. To compare the performance of the generated code with that of the original handwritten code, we ran both codes on a computer using different compilers. We found that the speed gain depends heavily on the quality of the compiler. We obtained performance gains of a factor of 3 and 1.2 over the handwritten code for the gfortran 4.1.2 and pathscale 3.0 compiler, respectively. This confirms that optimizations performed by CTADEL like common subexpression elimination are very effective. However, the use of common subexpression elimination results in a large number of temporary variables and hence increases the memory usage. To reduce the increased memory usage, CTADEL should be adapted so that in the generated code temporary variables are re-used.

In essence, we have shown that the application domain of CTADEL is not limited to finite difference methods, but, with small extensions, it is also suitable to generate efficient code for other discretization methods.

In Chapter 3, we optimized the parallel implementation of the HIRLAM weather forecast model. Parallelization in HIRLAM is achieved by domain decomposition. Each processor performs the calculations on a rectangular domain which we call the processor domain. Because of data dependence, data
communication between processors is required. As a result, the execution time of the parallel program involves communication time. In the current parallel implementation of HIRLAM, the communication is completely finished before the processor starts its calculation. Hence, the communication time is fully additional to the execution time. It has a significant negative impact on the efficiency of the parallel program. To reduce this impact, we start the calculation while the communication is still in progress. The purpose is to hide the communication time behind the calculation time. The advantage of overlapping communication with calculation has a counterweight in an increased calculation time arising from splitting the processor domain. The balance between gains and losses depends on the computer hardware. Therefore, we investigated several overlap strategies, ranging from one that may be the fastest on vector machines, to one that aims at maximum overlap, regardless the additional computational costs of overlapping. We found that if the interconnect is very fast (e.g., based on the Myrinet-MX protocol), the strategy designed for vector machines is the fastest. However, because the time spent in communication is small, the gains from overlapping are negligible. On the other hand, if the interconnect is not very fast (e.g., based on the Ethernet protocol), the savings are considerable. On our hardware, the increased calculation time arising from splitting the processor domain is small. Hence the strategy which has maximum overlap is the most efficient. In general, we concluded that depending on the network connection little to considerable savings on total execution times can be achieved by overlapping communications with calculations. On our hardware, the savings of a cheap interconnect are so substantial that there is no need for an expensive interconnecting network. However, if a fast (hence expensive) interconnecting network is available, the attainable savings are negligible, and then they surely do not warrant the associated code complications.

The above observations suggested that the HIRLAM system can be modified to achieve a higher performance, in particular on multiple clusters of a grid with a slow interconnect, or on a wide area computer. However, the deployment of an overlapping communication with calculation method in a production code encounters several issues. These are the problems posed by the global communication required in the implicit parts of the time stepping schemes, by the MPI buffer limitation, and more important how to cope with the required code complexities. We gave suggestions to resolve these problems. On the issue of global communication we can design an algorithm in which the calculations are started before all data needed to complete the calculations have been received. On the issue of the MPI buffer limitation, we suggested several solutions: we can configure the buffer size during installation of MPI; we can use the MPI buffered communication; we can reduce the message size by message splitting; we can apply the “halo on demand” technique to reduce the required communication; or we can use the MX protocol because this protocol allows overlap
without a limitation on the message size. The biggest challenge in implementing an overlapping communication with calculation method is the required code complexity. This problem can be resolved by using a code generation tool.

In Chapter 4 we extended CTADEL to generate parallel programs. We chose the Single Program Multiple Data as the parallel programming model. With this model, a parallel program consists of four steps: domain decomposition; data distribution; calculations and communications; and results collection. We defined templates to generate code for each step of a parallel program. An advantage of our technique is that the code generator automatically derives the size of the data to be communicated. Hence, the user does not have to worry about this.

By applying this technique we have successfully generated efficient parallel code for the Shallow-Water equations. Currently, the generated parallel program uses non-overlapping communication method. The technique needs to be improved to be able to generate parallel code in which communications are automatically overlapped with calculations.

Weather forecasting has seen many benefits from parallel computing. However, recent studies showed that simply increasing the large-scale parallelism will result in a poor performance for many scenarios where strong scaling is required. Therefore, in Chapter 5 we investigated a method to speed up the HIRLAM weather forecast model based on GPU computing. With a rapid development in performance in recent years, GPUs nowadays are being used as computational resources for many computationally intensive problems. We accelerated the dynamics routine of the HIRLAM weather forecast model by porting it on GPUs. In the dynamics routine the calculations on a certain grid point require information from horizontally neighboring grid points. In a conventional parallel program using distributed memory architecture, this data dependence results in requirements for the communication between the processors. The data dependence issue is resolved differently in CUDA. In CUDA the code that executes on the GPU is called the kernel. A kernel is executed by a number of threads which are batched into blocks. Consequently, synchronization would be required between blocks, but CUDA does not support synchronization between blocks. We investigated two methods to build independent blocks. The first is by recalculation: when a thread in a block needs information from a thread in a neighboring block, it calculates this information itself. The second is by splitting the dynamics routine into several kernels: a kernel then consists of blocks that can be executed independently. We found that the multi-kernel method is significantly more efficient than the recalculation approach. The performance of a CUDA program varies on the way that data is distributed to threads and the organization of threads into blocks. We enabled further optimizations by implementing the CUDA code with flexibilities in the number of grid points per thread and the number of threads per block.
From the experimental results we found that, assigning a vertical column of 32 grid points to each thread is the best. For the block size, we found that the larger the block, the more efficient the program is. In addition, we observed that distributing more threads to the first dimension of 3-dimensional blocks is much better. Because the CPU and the GPU cannot access memory of each other, the inputs and outputs of the program have to be transferred to and from the GPU before and after the calculations. The input/output transfer time is expensive and has negative impact on the performance of a CUDA program. We reduced the influence of the transfer time by using CUDA streams. A CUDA stream is a sequence of data transfers and/or kernel calculations that are executed in a specific order on a GPU. CUDA allows the use of multiple CUDA streams in parallel. We utilize this property of CUDA to overlap data transfers with kernel calculations. Specifically, we split the data transfers and kernel calculations into two CUDA streams so that one CUDA stream executes a kernel while another CUDA stream is transferring data. The results showed that by overlapping data transfers with kernel calculations we can reduce the total execution time with 36%. This way of overlapping reduces the overhead of data transfers by 63%.

Comparing to the C code on a state-of-the-art Intel CPU, the CUDA program on a single GPU has a speedup of 55. This speedup is much larger than the difference between the peak performance of the GPU and CPU that we used for experiments, which is around 20. This reflects the many optimizations that we did on the CUDA program, but it also indicates that the C program is far from optimal. Another reason is that the gcc compiler is not able to exploit the 4 cores with 8 threads of the CPU. It is also not unlikely that the GPU hardware matches the type of operations in dynamics better than the CPU hardware.

A single GPU has not enough memory for the production run. To run larger domains, it is possible to use the concept of grids, as defined by CUDA. In that case, the domain is split into subdomains which fit into the GPU memory, much like a parallel implementation. But the grids are processed serially. So after the GPU has processed a subdomain, the context is switched to the next grid. With each context switch, the whole GPU memory must be refreshed: the processed subdomain is transferred from the GPU to the CPU and the next subdomain is transferred from the CPU to the GPU. The data dependency issue is handled by transferring also halo zones. These transfers have to be done in every time step, which is very expensive. Alternatively, it would be possible to use more GPUs in parallel, each for one subdomain. We have explored this multiple GPUs option. On 2 and 4 GPUs, our experiments showed that the calculation times scale extremely well with the number of GPUs. For the data transfer times, the configuration with 2 GPUs has a speedup of nearly 2 over a single GPU. But the data transfer time in case of 4 GPUs stays almost equal to the case of a 2-GPU configuration. This results in a poor parallel speedup of 2.7
Chapter 7. An overview of research results achieved

for a 4-GPU configuration.

The results indicated that the use of GPUs for weather forecasting is promising. However, before operational weather forecasting can make use of GPUs, the other compute intensive parts of the HIRLAM forecast model would also have to be converted to GPU-enabled code. This refers in particular to the physics package. In the physics, the calculations on a certain grid point only depend on the vertically neighboring grid points. Hence, we can assign one thread to each vertical column of grid points. As a result, the block synchronization and the communication in case of multiple GPUs/CPUs issue is relatively minor.

The observations have convinced us that it is useful to extend the CTADEL code generation tool to generate code for GPUs. The extension needs not only to be limited to generate a simple CUDA program which consists of the kernel and the host code. The code generator should also be able to generate the CUDA stream program in which the data transfer is hidden behind the kernel execution as much as possible, for a single or multiple GPUs configuration.

In Chapter 6 we showed our extension to CTADEL to automatically generate highly efficient CUDA codes. At first, we extended CTADEL to generate simple CUDA programs. We defined templates to generate the kernel and host code of a CUDA program. Next, to have overlap between data transfers and kernel calculations, we adapted CTADEL to generate CUDA stream programs. Our technique consists of four steps. Firstly, CTADEL generates the calculation streams, which are lists that reflect the invocation order of the kernels. Secondly, the data transfers are added to overlap with the kernel calculations. The output of this step is the CUDA stream code. To avoid confusion, we note that a CUDA stream code is the host code of a CUDA 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 stream code, these invocations are organized into two CUDA streams which are executed in parallel. We proposed two approaches to generate the CUDA stream codes. In the first approach, the calculation of a kernel (the current kernel) is overlapped with the transfer of the inputs of the next kernel (the kernel to be invoked after the current kernel) and the outputs of the previous kernel (the kernel that was invoked before the current kernel). In this method, it can happen that the execution time of a kernel is much larger than the time to transfer the inputs/outputs which are overlapped with that kernel, whereas in another overlap phase, the transfer time is larger than the kernel execution time. If this is the case, we do not achieve the maximum profit of overlapping. In the second approach, we balance the data transfer time with the kernel execution time. We add more transfers to overlap with an expensive kernel and remove transfers if the kernel execution time is smaller than the transfer time.

In the third step, CTADEL derives the theoretical time of all CUDA stream codes based on the transfer time of inputs/outputs and the calculation time of
kernels. The kernel calculation 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. The theoretical time of a CUDA stream code is an aggregation of the transfers and execution time in case of non overlap, or is equal to the maximum value of those two times if there is overlap between a kernel execution and data transfers. Finally, CTADEL chooses the CUDA stream code that has the smallest theoretical time to generate the optimal CUDA stream program.

We applied the generation of CUDA stream programs to the dynamics routine of the HIRLAM weather forecast model. Our experiments showed that the CUDA stream code generation approach in which the data transfer time is balanced with the kernel calculation time is more efficient than the approach in which this balance is not considered. Hence, we used this method to generate the optimal CUDA stream program for the dynamics routine. The experimental results showed that the total execution time of the optimal CUDA stream program is 38% smaller than that of the CUDA program in which the data transfers are not overlapped with the kernel calculations.

Currently, the generated CUDA program is limited to a single GPU implementation. To have a wider application, the technique should be extended to generate CUDA program for multiple GPUs and/or multiple CPUs configuration.

Several options could be considered to improve the performance of HIRLAM, options that hitherto were out of the question because they require complex coding, or that were never considered at all because it was anticipated that the possible savings did not warrant the required amount of recoding. For some of those options we have shown that they may give considerable savings. It hence is worth to extend the code generator to implement those options while at the same time avoiding the burden of having to write and maintain complex codes. We showed this for overlapping communications and computations, and for several forms of GPU implementations. For the latter, we actually carried out the extensions to the code generator CTADEL.