# Towards Tangent-linear GPU Programs using OpenACC

Bui Tat Minh
The Sirindhorn International Thai-German Graduate School of Engineering (TGGS)
King Mongkut's University of Technology North Bangkok (KMUTNB)
Bangkok, Thailand
minh.bui@rwth-aachen.de

Michael Förster, Uwe Naumann
LuFG Informatik 12: Software and Tools for Computational Engineering
RWTH Aachen University
Aachen, Germany
{foerster,naumann}@stce.rwth-aachen.de

## ABSTRACT

Recently, Graphics Processing Units(GPUs) have emerged as a very promisingly powerful resource in scientific computing. Algorithmic Differentiation is a technique to numerically evaluate first and higher derivatives of a function specified by a computer program efficiently up to machine precision. Derivative programs which are used to compute derivatives of functions are so-called tangent-linear program and adjoint program. This paper aims to offload any particular independent loop in tangent-linear program to GPUs. The proposed technique is OpenACC APIs for annotating an independent loop to be executed in parallel on GPUs. Our case study for OpenACC tangent-linear code shows an enormous speedup. OpenACC shows its simplicity of accelerating tangent-linear code by hiding the data movement between CPU and GPU memory.

## Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming, Parallel programming; G.1.4 [**Quadrature and Numerical Differentiation**]: Automatic differentiation

## General Terms

Performance, Algorithms

## Keywords

arithmetic differentiation, tangent-linear model, data parallelism, SIMD, OpenACC

## 1. INTRODUCTION

In simulation science, the sensitivity of an output of a simulation program with respect to its input change plays an important role. Scientists would like to know how much the output values change when the input values are perturbed. If these changes are big, the scientists should pay attention to the accuracy of their mathematical model. An inaccurate model leads to wrong simulation results. Therefore, achieving sensitivity in simulation science becomes crucial.

Algorithmic (also Automatic) Differentiation, or AD for short, is a technology for computing sensitivities by automatically augmenting computer programs with additional statements of derivative computation. AD exploits the fact that even the most complicated program still executes a sequence of elementary operations (plus, subtract,division, product, etc) or common intrinsic functions (sine, cosine, exp, etc). Any complicated program can be augmented with statements to compute derivatives of these operations and functions. The augmented program is so-called derivative program. Within a first-order derivative program, both function values and first-order derivatives are computed. By re-application of AD onto the first-order derivative program, we obtain derivative code to compute second-order derivatives and so on. Derivative computation is computer-aid calculation; therefore, AD avoids truncation errors which we usually encounter using Finite Differences[6].

In November 2011, a group of companies comprising CAPS, CRAY, NVIDIA, and PGI announced a new standard for GPU computing, called OpenACC APIs. OpenACC provides directives, runtime library routines, and environment variables to programmer for offloading code in C/C++ and Fortran programs from the CPU to an attached accelerator device, such as GPU. The technique that OpenACC compiler uses is source transformation. OpenACC compiler such as the commercial PGI compiler transforms OpenACC augmented code into highly optimized CUDA code. In March 2013, PGI compiler officially supported C++ in pgc++ version 13.2. This brings us a good chance to experiment with AD code on the GPUs.

*This paper aims to offload any particular independent loop in tangent-linear code to GPUs. A case study is conducted to measure speedup change with respect to loop size parameters in a parallel tangent-linear program.*

The paper is organized in the following structure. Section 2 discusses related work to this paper. Section 3 introduces us to a basic knowledge of Tangent-linear Model, Derivatives Code Compiler and OpenACC. Section 4 proposes OpenACC tangent-linear code, followed by a case study in Section 5. Section 6 is our conclusion and outlook of this paper.

## 2. RELATED WORK

In the paper *Toward adjoint OpenMP* [2], the authors discussed parallelized adjoint code using OpenMP. The authors experimented with derivative code on multi-core CPUs. OpenACC APIs is a new approach to offloading code to

*SoICT '13*, December 05 - 06 2013, Danang, Viet Nam
Copyright 2013 ACM 978-1-4503-2454-0/13/12...$15.00.
http://dx.doi.org/10.1145/2542050.2542059

GPUs. The programing paradigm is very close to OpenMP, where we annotate parallel sections with pragmas. But OpenACC works with accelerator devices, like GPUs, which have separated memory space from the CPUs. OpenACC hides the complexity of moving data back and forth between host memory and device memory.

## 3. BACKGROUND

### 3.1 Tangent-linear Model

Consider a multivariate function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$y = F(x),$$

$F$ is assumed to be continuously differentiable in $\mathbb{R}^n$. This leads to the existence of the Jacobian matrix of $F$.

*Definition 1.* The Jacobian matrix of $F$ [6]:

$$\nabla F(x) \equiv \left(\frac{\partial y_j}{\partial x_i}\right)_{i=0,\dots,n-1;j=0,\dots,m-1} \in \mathbb{R}^{m\times n}$$

Each entry of the Jacobian matrix is a partial derivative of one output component $y_j$ of the output vector $y$ with respect to one input component $x_i$ of the input vector $x$.

*Definition 2.* The tangent-linear model $F^{(1)}$[6] $: \mathbb{R}^{m\times n} \times \mathbb{R}^n \rightarrow \mathbb{R}^m$ is defined as

$$y^{(1)} = \nabla F(x) \cdot x^{(1)}$$

We call $x^{(1)}$ a tangent-linear direction and $y^{(1)}$ a directional derivative in that direction. Ranging $x^{(1)}$ over the Cartesian basis vectors in $\mathbb{R}^n$ yields the *Jacobian matrix.* The procedure of assigning $x^{(1)}$ to a basic vector, so-called *seed vector*, is called *seeding.* Calculating $y^{(1)}$ is called *harvesting* in tangent-linear model.

At the end of executing the tangent-linear model, we obtain both values and derivatives of function $F$.

*Example 1.* Given:

$$y = F(x) : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 2(x_0^2 + x_1^2) \\ sin(x_0^2 + x_1^2 + \frac{\pi}{2}) \end{pmatrix} \tag{1}$$

Find $\frac{\partial(y_0)}{\partial(x_1)}$ and $\frac{\partial(y_1)}{\partial(x_1)}$ at $\breve{x} = (\sqrt{\frac{\pi}{2}}, \sqrt{\frac{\pi}{2}})$ ?

We consider the seed vector: $x^{(1)} = (x_0^{(1)}, x_1^{(1)}) = (0, 1)$. In the tangent-linear model, the following values are calculated as in Table 1.

### 3.2 Derivatives Code Compiler

Derivatives code compiler, or dcc for short, transforms a numerical program written in a subset of C/C++ into a derivative program using a source-to-source transformation technique. Dcc is developed at the LuFG Informatik 12: Software and Tools for Computational Engineering (STCE), RWTH Aachen University. Applying a set of techniques of AD, dcc generates two versions of derivative code: tangent-linear code by forward mode AD or adjoint code by reverse mode AD[6].

The input of dcc is a numerical program written in a subset of C/C++. For the list of syntax accepted by dcc, refer

| Local tangent-linear model |
|---|
| $v_0^{(1)} = x_0^{(1)} = 0$ <br> $v_0 = x_0 = \sqrt{\frac{\pi}{2}}$ |
| $v_1^{(1)} = x_1^{(1)} = 1$ <br> $v_1 = x_1 = \sqrt{\frac{\pi}{2}}$ |
| $v_2^{(1)} = \frac{\partial\varphi_2}{\partial v_0} \cdot v_0^{(1)} + \frac{\partial\varphi_2}{\partial v_1} \cdot v_1^{(1)}$ <br> $= 2v_0 \cdot v_0^{(1)} + 2v_1 \cdot v_1^{(1)} = 2 \cdot \sqrt{\frac{\pi}{2}} \cdot 0 + 2 \cdot \sqrt{\frac{\pi}{2}} \cdot 1 = \sqrt{2\pi}$ <br> $\varphi_2 = v_2 = v_0^2 + v_1^2 = (\sqrt{\frac{\pi}{2}})^2 + (\sqrt{\frac{\pi}{2}})^2 = \pi$ |
| $v_3^{(1)} = \frac{\partial\varphi_3}{\partial v_2} \cdot v_2^{(1)} = 2 \cdot v_2^{(1)} = 2\sqrt{2\pi}$ <br> $\varphi_3 = v_3 = 2v_2 = 2\pi$ |
| $v_4^{(1)} = \frac{\partial\varphi_4}{\partial v_2} \cdot v_2^{(1)} = cos(v_2 + \sqrt{\frac{\pi}{2}}) \cdot v_2^{(1)} = cos(\pi + \frac{\pi}{2}) = 0$ <br> $\varphi_4 = v_4 = sin(v_2 + \frac{\pi}{2}) = sin(\pi + \frac{\pi}{2}) = -1$ |
| $y_0^{(1)} = v_3^{(1)} = 2\sqrt{2\pi}$ <br> $y_0 = v_3 = 2\pi$ |
| $y_1^{(1)} = v_4^{(1)} = 0$ <br> $y_1 = v_4 = -1$ |

Table 1: An example of local tangent-linear code.

The computation of $F(x)$ executes a sequence of elementary operations and functions $(\varphi_2, \varphi_3, \varphi_4)$. We augment the original code with derivative calculation before each elementary statement. We put local tangent variables $v_2^{(1)}$, $v_3^{(1)}$, and $v_4^{(1)}$ before their respective statements for computing $v_2$, $v_3$, and $v_4$ (so-called Single Assignment Code or SAC). At the end, we obtain both the function values and partial derivatives with respect to $x_1$ at $\breve{x} = (\sqrt{\frac{\pi}{2}}, \sqrt{\frac{\pi}{2}})$. They are $\frac{\partial(y_0)}{\partial(x_1)} = 2\sqrt{2\pi}$ and $\frac{\partial(y_1)}{\partial(x_1)} = 0$. Partial derivatives can also be computed using adjoint model. Please refer to [5] for adjoint example.

to [6]. The output of dcc is a derivative code, either tangent-linear code or adjoint code. We consider the following general case to understand what the input and output of dcc are:
Given:

$$F : \mathbb{R}^n \to \mathbb{R}^m, y = F(x), \tag{2}$$

where $x$ and $y$ are not aliased (represented by different program variables). The implementation of $F$, which is the input of dcc, has the following signature:

```
void f(int n, int m, double *x, double *y).
```

The tangent-linear program:

$$F^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m : \begin{pmatrix} y \\ y^{(1)} \end{pmatrix} = F^{(1)}(x, x^{(1)}) \tag{3}$$

computes both $y$ and $y^{(1)}$ and has a signature as follows:

```
void t1_f(int n, int m, double *x, double* t1_x,
                          double *y, double *t1_y).
```

where `t1_` denotes a tangent-linear subroutine and tangent-linear variables. This program is the output of dcc. We need to seed vector `t1_x` for $n$ times to accumulate the whole Jacobian matrix.
Not only can dcc generate tangent-linear code of any computer program but also adjoint code. Please refer to [6] for explanation about adjoint code. This paper focuses on porting independent loops in tangent-linear code to the GPUs.

## 3.3 OpenACC

OpenACC fits our goal to offload independent loops in tangent-linear code to GPUs. OpenACC provides some features, namely accelerator initialization, data movement, and kernel creation.

### 3.3.1 Accelerator initialization

OpenACC compiler helps to initialize the accelerator before executing the offloaded code. For example, NVIDIA usually puts the GPUs into idle mode when they are not used for either graphics rendering or general-purpose computation. There is a small overhead in initializing the GPUs to active mode. OpenACC provides an initialization routine, called *acc_init()*, to activate the accelerator from idle mode.

### 3.3.2 Data construct

OpenACC compiler takes care of data movement implicitly. Currently, GPU memory is not mapped into the host's virtual memory space. OpenACC provides *data construct*, followed by *data clauses* to allocate device variables *(create)*, copy data from the host to the accelerator at region entry and back to the host at region exit *(copy)*, or only copy data over from the host to the accelerator *(copyin)* or reverse *(copyout)*. The full list of data clauses can be found in [1]. The following example illustrates the data construct:

```
#pragma acc data pcopyin(A[0:n])
{
structure block
}
```

The PGI compiler generates highly optimized CUDA code which handles array A on the device side. The *pcopyin* clause is an extension of *copyin* clause. The *pcopyin* is the abbreviation of *present or copyin*, in which if A is not present in the accelerator memory, A is allocated and copied from the host to the accelerator at region entry, as with the *copyin* clause. We do not have to specify the data type of A. Upon the exit of the region, A is deallocated. This approach contrasts with verbose *cudaMemcpy* commands in CUDA. All these procedures are performed automatically.

### 3.3.3 Accelerator Compute Constructs

There are two pragmas which help us to create a kernel. We can use the *parallel* or the *kernels* pragma to annotate independent loops. There are two significant differences between them; the *parallel* pragma creates only one CUDA kernel, whereas the *kernels* pragma turns independent loops inside it to kernels. The other difference between two constructs is the user-involvement in kernel scheduling. With the *kernels* construct, the compiler is free to map iterations to a parallelism level and choose an appropriate number of threads and blocks. On the other hand, the *parallel* construct allows the user to choose kernel scheduling manually.

```
#pragma acc parallel{
compute region
}
```

or

```
#pragma acc kernels{
compute region
}
```

A detailed discussion of *parallel* and *kernels* can be found in PGI Insider[3].

## 4. OPENACC TANGENT-LINEAR CODE

In this section, we focus on OpenACC augmented tangent-linear code. Firstly, we consider a class of problem, which features data parallelism. The tangent-linear version of this problem class is also a data parallelism problem. Secondly, we propose OpenACC version for the tangent-linear version of this problem class.
We consider a multivariate function in equation (2) in Section 3.2. We assume that the problem class is a highly data-parallel problem, in which each component of output vector y can be calculated simultaneously. We have $m$ independent iterations which can be executed in parallel. A comparison between the implementation of $F$ and its tangent-linear is discussed below:

**Algorithm 1:** Implementation of $F$.

```
Data: x
Result: y
for $j \leftarrow 0$ to $m - 1$ do
    compute y[j];
end
```

In the tangent-linear version, the loop structure of the program is the same as in the original implementation of $F$. The tangent-linear version is built by augmenting auxiliary variables to the function implementation as we discussed in Section 3.1. Therefore, when the $F$ implementation can be executed in parallel, its tangent-linear can be executed in

**Algorithm 2:** The tangent-linear version of $F$.

```
Data: x, t1_x /*additional input t1_x          */
Result: y, t1_y /*additional output t1_y       */
for $j \leftarrow 0$ to $m-1$ do
    compute t1_y[j] and y[j]; /*derivative t1_y[j] */
end
```

parallel in the same way.
In order to offload a program with data parallelism to GPUs, we need to clarify two requirements: data movement and kernel scheduling.
Firstly, we need to identify variables, which need to be copied over from host memory to device memory, or vice versa. We achieve this requirement with the *data* pragma. In tangent-linear model, apart from the input and output of the independent loop, x and y, respectively, there are new input and output variables, namely tangent variables t1_x and t1_y.
Secondly, we need to assign the loop to the hardware parallelism level.
With NVIDIA GPUs, PGI compiler maps data parallelism to two levels of hardware parallelism: threads and blocks. In Algorithm 3, we use the *parallel* pragma to annotate the following loop as a kernel. We use the *loop* pragma to inform the compiler that this is an independent loop. Furthermore, the *vector* data clause maps iterations to threads in GPUs. Up to this point, the exact number of the GPU threads that we use is still not defined. The PGI compiler will find the appropriate resources to perform this kernel. The exact number of blocks per grid and threads per block depends on the availability of hardware, such as the number of required registers per thread. For example, with NVIDIA Quadro 6000 there are 32768 registers per block. In our approach, we trust the compiler to choose the best GPU resources.

**Algorithm 3:** OpenACC tangent-linear version

```
Data: x, t1_x
Result: y, t1_y
#pragma acc data copyin x and t1_x, copy y and t1_y
    #pragma acc parallel
        #pragma acc loop vector
            for $j \leftarrow 0$ to $m-1$ do
                compute t1_y[j] and y[j];
            end
```

For real-world problems which feature data parallelism, we have different loop structures apart from the case above. However, the tangent-linear version of these problems has the same parallelism structure as in the original function implementation. Therefore, when the function implementation is a data parallelism problem, we can parallelize the corresponding tangent-linear version.
We will experiment with the OpenACC tangent-linear version of a case study in the next section.

## 5. CASE STUDY

Given $X \in \mathbb{R}^{n\times p}$, a constant matrix $A \in \mathbb{R}^{p\times m}$ and $Y \in \mathbb{R}^{n\times m}$
$Y = F(X) : \mathbb{R}^{n\times p} \rightarrow \mathbb{R}^{n\times m}$ that:

$$y_{i,j} = \begin{cases} \sum_{k=1}^{p} (x_{i,k} \times a_{k,j}) + \sin(y_{i-1,j-1}) & \text{if } i,j > 1 \\ \sum_{k=1}^{p} (x_{i,k} \times a_{k,j}) \quad \text{if } i = 1 \vee j = 1 \end{cases}$$

in which $1 \leq i \leq n$ and $1 \leq j \leq m$

One possible implementation of $Y = F(X)$ is shown in Listing 1, in which the calculation of $Y$ is decomposed into two parts: matrix multiplication procedure, called g function and re-computation procedure of $Y$, called f function. In Function g, each element of output matrix $Y$ is a dot

```
1  void g(int n, int m, int p, double** x, ↪
       ↩ double** A, double** y)
   #pragma ad indep x
3  #pragma ad dep y
   {
5    int i = 0;
     int j = 0;
7    int k = 0;
     for ( i = 0; i < n; i++ ){
9      for ( j = 0; j < m; j++ ){
         for ( k = 0; k < p; k++ ){
11         y[i][j]=y[i][j]+ x[i][k]*A[k][j];
         }
13     }
     }
15 }
   void f(int n, int m, int p, double** x, ↪
       ↩ double** A, double **y)
17 #pragma ad indep  x
   #pragma ad dep y
19 {
     int i = 0;
21   int j = 0;
     g( n, m, p, x, A, y );
23   for( i = 1; i < n; i ++ ){
       for( j = 1; j < m; j++ ){
25       y[i][j]=y[i][j] + sin(y[i-1][j-1]);
       }
27   }
   }
```

Listing 1: Function implementation of casestudy

product of one vector, which is a row in $X$, and one vector, which is a column in $A$. The calculation of each element in $Y$ is totally independent from the other elements. Therefore, in g all elements of output matrix $Y$ can be computed simultaneously. In this analysis, two outer loops (line 8 and line 9) can be parallelized easily. The inner-most loop (line 10) which performs the reduction of $y_{i,j}$ can be parallelized but will not be further discussed in this paper.

In Function f, the values of each element in matrix $Y$, except those in the first row and the first column, are recomputed using its current value and the left-hand side value on its diagonal. Function f calls function g in line 22, before re-computing matrix $Y$ in line 25. Intuitively, the value of one element depends not only on its current value but also on the value of the previous on-diagonal element. This paper will consider the two outer loops of f (line 23 and 24) as dependent loops and will not try to parallelize them on GPUs.

Our loop dependency analysis of g results in a possible OpenACC version of g in Listing 2.

```
   #pragma acc parallel ↪
       ↩ pcopyin(x[0:n][0:p], A[0:p][0:m]) ↪
       ↩ copy(y[0:n][0:m])
2  {
    #pragma acc loop gang
4   for( i=0; i<n; i++ ){
      #pragma acc loop vector
6     for( j=0; j<m; j++ ){
        for( k=0; k<p; k++ ){
8         y[i][j] = y[i][j] + x[i][k] * ↪
              ↩ A[k][j];
        }
10    }
    }
12 }
```

Listing 2: The OpenACC version of g

Two adjacent pragmas *data* and *parallel* can be combined into one *parallel* pragma, where data movement is denoted by data clauses. Data is copied into device memory with *pcopyin* or copied to/from host memory with *copy*. The i loop in line 4 is scheduled to grid, whilst the j loop in line 6 is scheduled to block.

The loop dependence analysis of f and g brings us a draft view of the further step to porting tangent-linear code to the GPUs. In our experiment, we use dcc-v1.0 to generate a tangent-linear version of g, called t1_g. Base on the loop dependency analysis of g, we know the loop structure of t1_g. Then, we augment t1_g with OpenACC pragmas for our experiment.
For explanation, we only introduce hand-written tangent-linear code of g. The OpenACC dcc-generated tangent-linear version t1_g_acc and t1_f_acc is present in Appendix. Listing 3 shows the first OpenACC tangent-linear code while Listing 4 optimizes data movement between host memory and device memory.

```
   #pragma acc parallel ↪
       ↩ pcopyin(x[0:n][0:p], A[0:p][0:m], ↪
       ↩ t1_x[0:n][0:p])  ↪
       ↩  copy(y[0:n][0:m], t1_y[0:n][0:m])
2  {
    #pragma acc loop gang
4   for( i=0; i<n; i++ ){
      #pragma acc loop vector
6     for( j=0; j<m; j++ ){
        for( k=0; k<p; k++ ){
8         t1_y[i][j] = t1_y[i][j] + ↪
              ↩ t1_x[i][k] * A[k][j];
          y[i][j] = y[i][j] + x[i][k] * ↪
              ↩ A[k][j];
10      }
      }
12  }
   }
```

Listing 3: The first OpenACC tangent-linear version. Zero matrices y and t1_y are copied over from host memory to device memory via *copy* clause (line 1)

In our OpenACC tangent-linear version t1_g_acc, y and t1_y need not to be copied over from host memory to device memory. We would like these variables allocated and initialized to zero in device memory. Unfortunately, at the moment there is no OpenACC pragma or API which can help us achieve this [4]. Therefore, we change the code manually for efficiency. We allocate y and t1_y with the *create* pragma. Then, we initialize them to zero inside the loop body.

```
1  #pragma acc parallel ↪
       ↩ pcopyin(x[0:n][0:p], A[0:p][0:m], ↪
       ↩ t1_x[0:n][0:p]) ↪
       ↩  create(y[0:n][0:m], t1_y[0:n][0:m])  ↪
       ↩  copyout(y[0:n][0:m], t1_y[0:n][0:m])
   {
3     #pragma acc loop gang
      for ( i=0; i<n; i++ ) {
5     #pragma acc loop vector
        for ( j=0; j<m; j++ ) {
7         y[i][j]=0;
          t1_y[i][j]=0;
9         for ( k=0; k<p; k++ ) {
            t1_y[i][j]=t1_y[i][j] + ↪
                ↩ t1_x[i][k]*A[k][j];
11          y[i][j]=y[i][j] + x[i][k]*A[k][j];
          }
13      }
      }
15 }
```

Listing 4: Optimized OpenACC tangent-linear version for efficient data movement. We allocate y and t1_y in device memory via *create* clause (line 1). We later initialize y and t1_y to zero inside kernel function as in line 7 and 8

The optimized data movement helps us reduce unnecessary data traffic between host and device memory. The following table shows improvement in runtime performance.

| Experimental cases at device memory | Copyin | Kernel | Copyout |
|---|---|---|---|
| Copy zero matrices | 40,890 $\mu$s | 210,799 $\mu$s | 16,420 $\mu$s |
| Initialization zero matrices | 24,498 $\mu$s | 210,556 $\mu$s | 16,501 $\mu$s |

Table 2: Data clause profiling.

We profile runtime performance of kernel execution and data movement for two cases with $n = m = p = 1024$. It is clear that when we initialize matrices y and t1_y in device memory, we reduce the traffic over the PCI express bus between device memory and host memory. This results in the *copyin time* reduces to 41%.
With the optimized data movement in the tangent-linear version, we determine how speedup varies with respect to problem size in Algorithm 4.

We experiment with 27 cases, in which each parameter takes one of three values 1024, 2048, and 4096. The speedup is the ratio of CPU time to GPU time. The runtime performance of the whole application t1_f and t1_f_acc (see Appendix) is taken in account. The experiment is conducted with Intel Xeon X5650 EP, GPU NVIDIA QUADRO 6000, and OpenACC-supported PGI compiler (pgc++13.2).

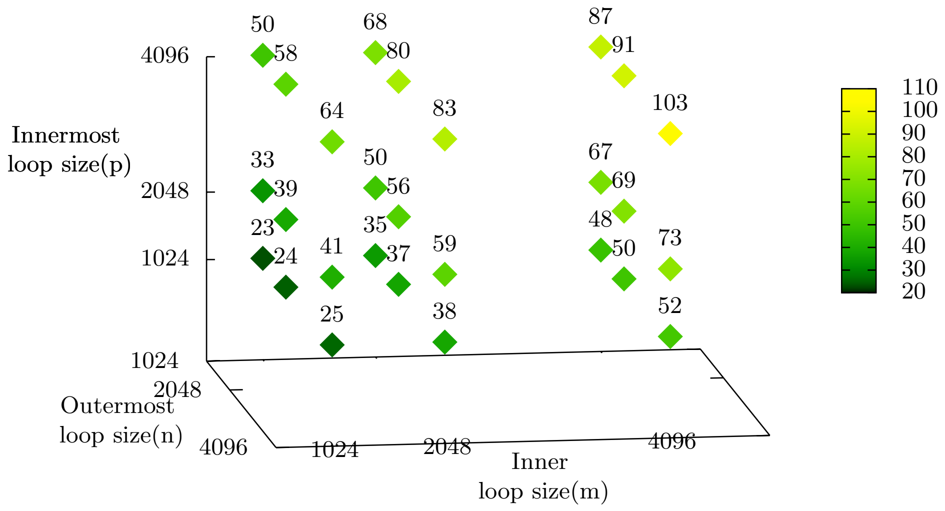

Figure 1: Speedup vs loop size parameter.

**Algorithm 4:** Speedup measurement procedure.

```
Data: x, t1_x, A
Result: CPU and GPU Runtime, Speedup
seeding t1_x;
for i in 1024, 2048, 4096 do
    for j in 1024, 2048, 4096 do
        for k in 1024, 2048, 4096 do
            profile t1_f on CPU;
            profile t1_f_acc on GPU;
            y_err = check(y, y_acc);
            t1_y_err = check(t1_y, t1_y_acc);
            if y_err or t1_y_err then
                exit(1);
            else
                calculate speedup;
                write CPU time, GPU time, speedup to
                file;
            end
            reset y, y_acc, t1_y, t1_y_acc;
        end
    end
end
```

The 4D plot in Figure 1 shows the relationship between speedup and problem size. We achieve the best speedup of 103x when $n = m = p = 4096$. When $n = m = p = 1024$, we obtain the lowest speedup of 23x. The experiment shows speedup is directly proportional to each loop size parameter. There are two possibilities to create computationally intensive code in the tangent-linear version `t1_g`. Firstly, we increase $n$ or $m$ to put more loop iterations on GPUs. Secondly, we increase $p$ to put more work on each j iteration. In both cases, we achieve higher speedup when we increase either $n$, $m$, or $p$. GPUs are suitable for computationally intensive code. The intensive code is a loop with a big number of independent iterations and huge workload in each iteration.

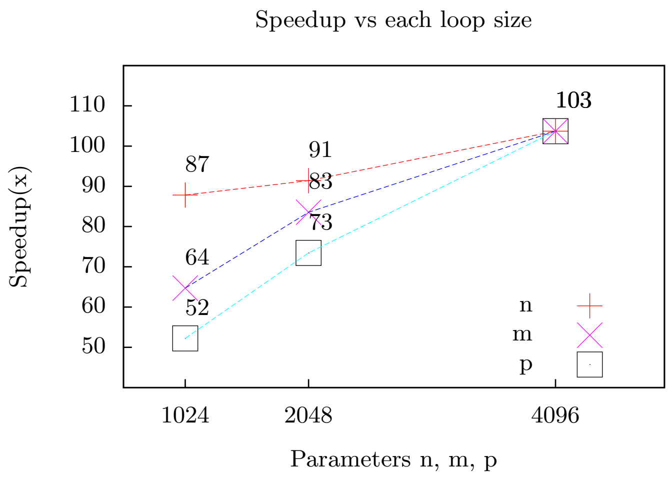

Figure 2: Speedup vs different parameter n, m, and p.

In Figure 2, we find which loop size parameter influences the speed up the most. Each loop size parameter increases from 1024 to 4096. At each point, we record the highest possible speedup. For example, when $n = 1024$, we have the highest speedup of 87x, only when $m = 4096$ and $p = 4096$. When $p$ increases from 1024 and 4096, the highest speedup increases from 52x to 103x respectively (approximately two times). The speedup rises sharply as p increases in our case study. In our case study, $p$ is more important in driving the speedup forward. This implies that if we put more workload into each iteration, the speedup increases faster than in the case where we increase the number of iterations.

## 6. CONCLUSIONS

Our case study has shown the simplicity of offloading computationally intensive parts of tangent-linear code to the GPUs with OpenACC. OpenACC hides data movement between the host memory and the device memory. We also interpret the phrase *'computationally intensive code'* as an independent loop with a huge number of iterations (thousands or millions) and large amount of work in each iteration.

One weak point of OpenACC is its lack of support for any debugging tool. Compiler pgc++ version 13.2 does not provide any debugging tool. Nevertheless, OpenACC is very promising not only for derivative programs but also for other scientific programs which feature data parallelism. OpenACC is expected to be integrated to OpenMP in the near future. The initialization of variables in device memory is necessary. From the case study, we expect OpenACC specification should have pragmas which allow the user to initialize data in device memory instead of copying over from host memory.

This paper explored the possibility of offloading intensive parts of tangent-linear code to GPUs. From our approach and case study, we can infer that tangent-linear code retains the same block structure as that of the original implementation of the function. The speedup of the parallel tangent-linear code is directly proportional to the loop size parameter. We aim to extend dcc to generate parallel tangent-linear code on the GPUs.

## 7. ACKNOWLEDGEMENT

We would like to thank Ekkapot Charoenwanit for proofreading this paper many times. We would like to give special thanks to the HPC team at RWTH Aachen University for providing both GPUs hardware and commercial PGI compiler. This work was supported by grants from the German Academic Exchange Service and the German Federal Foreign Office.

## 8. REFERENCES

[1] The OpenACC ™ Application Programming Interface version 1.0, November 2011.

[2] M. Förster, U. Naumann, and J. Utke. Toward Adjoint OpenMP. Technical Report AIB-2011-13, RWTH Aachen, July 2011.

[3] T. P. Group. OpenACC Kernels and Parallel Constructs. `http://www.pgroup.com/lit/articles/insider/v4n2a1.htm`, August 2012. [Online; accessed 29-July-2013].

[4] T. P. Group. Userforum: Initialize global variables with OpenACC pragma. `www.pgroup.com/userforum/viewtopic.php?t=3869`, May 2013. [Online; accessed 03-August-2013].

[5] B. T. Minh. Tangent-Linear and Adjoint GPU Code. diploma thesis, The Sirindhorn International Thai-German Graduate School of Engineering, King Mongkut's University of Technology North Bangkok, May 2013.

[6] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012.

## APPENDIX

## A. OPENACC TANGENT-LINEAR VERSION OF F AND G FUNCTION

```
 1  void t1_g_acc(int n, int m, int p, ↪
      ↩ double** x, double** t1_x, ↪
      ↩ double** A, double** y, double** ↪
      ↩ t1_y)
    #pragma ad indep x t1_x
 3  #pragma ad dep y t1_y
    {
 5    int i=0;
      int j=0;
 7    int k=0;
      double v1_0=0;
 9    double t1_v1_0=0;
      double v1_1=0;
11    double t1_v1_1=0;
      double v1_2=0;
13    double t1_v1_2=0;
      double v1_3=0;
15    double t1_v1_3=0;
      double v1_4=0;
17    double t1_v1_4=0;
      #pragma acc parallel ↪
        ↩ pcopyin(x[0:n][0:p], ↪
        ↩ A[0:p][0:m], t1_x[0:n][0:p]) ↪
        ↩ create(y[0:n][0:m], ↪
        ↩ t1_y[0:n][0:m])  ↪
        ↩ copyout(y[0:n][0:m], ↪
        ↩ t1_y[0:n][0:m])
19    {
      #pragma acc loop gang
21    for ( i=0; i<n; i++ ) {
        #pragma acc loop vector
23      for ( j=0; j<m; j++ ) {
          y[i][j] = 0;
25        t1_y[i][j] = 0;
          for ( k=0; k<p; k++ ) {
27          t1_v1_0=t1_y[i][j];
            v1_0=y[i][j];
29          t1_v1_1=t1_x[i][k];
            v1_1=x[i][k];
31          t1_v1_2=0;
            v1_2=A[k][j];
33          t1_v1_3=v1_2*t1_v1_1;
            v1_3=v1_1*v1_2;
35          t1_v1_4=t1_v1_0+t1_v1_3;
            v1_4=v1_0+v1_3;
37          t1_y[i][j]=t1_v1_4;
            y[i][j]=v1_4;
39        }
        }
41    }
      }
43  }
```

Listing 5: The OpenACC tangent-linear version t1_g_acc

In tangent-linear version of g, auxiliary variables from line 5 to line 17 are used to store intermediate values for derivative computation. The auxiliary variables calculation is localized to each thread from line 27 to line 36. These variables are stored in register memory of GPUs for low latency access. The OpenACC compiler has to choose the appropriate number of threads, number of blocks to fit the requirement of number of needed registers per thread.

```
 1  void t1_f_acc(int n, int m, int p, ↪
      ↩ double** x, double** t1_x, ↪
      ↩ double** A, double** y, double** ↪
      ↩ t1_y)
    #pragma ad indep x t1_x
 3  #pragma ad dep y t1_y
    {
 5    int i=0;
      int j=0;
 7    double v1_0=0;
      double t1_v1_0=0;
 9    double v1_1=0;
      double t1_v1_1=0;
11    double v1_2=0;
      double t1_v1_2=0;
13    double v1_3=0;
      double t1_v1_3=0;
15    t1_g_acc(n,m,p,x,t1_x,A,y,t1_y);
      for ( i=1; i<n; i++ ) {
17      for ( j=1; j<m; j++ ) {
          t1_v1_0=t1_y[i][j];
19        v1_0=y[i][j];
          t1_v1_1=t1_y[i-1][j-1];
21        v1_1=y[i-1][j-1];
          t1_v1_2=cos(v1_1)*t1_v1_1;
23        v1_2=sin(v1_1);
          t1_v1_3=t1_v1_0+t1_v1_2;
25        v1_3=v1_0+v1_2;
          t1_y[i][j]=t1_v1_3;
27        y[i][j]=v1_3;
        }
29    }
    }
```

Listing 6: The OpenACC tangent-linear version t1_f_acc

The tangent-linear version of f is executed on the CPU side. f calls parallel tangent-linear version t1_g_acc in line 15.