AD Master Class 4

Pushing performance using SIMD vectorization

Viktor Mosenkis
`viktor.mosenkis@nag.co.uk`

20 August 2020

Experts in numerical algorithms
and HPC services

AD Masterclass Schedule and Remarks

- **AD Masterclass Schedule**
  - 30 July 2020 | Why the need for Algorithmic Differentiation?
  - 6 August 2020 | How AD works
  - 13 August 2020 | Testing and validation
  - 20 August 2020 | Pushing performance using SIMD vectorization
  - 27 August 2020 | Bootstrapping validated adjoints on real-world codes

- **Remarks**
  - Please submit your questions via the questions panel at any time during this session, these will be addressed at the end.
  - A recording of this session, along with the slides will be shared with you in a day or two.

Dialogue

We want this webinar series to be interactive (even though it's hard to do)

- We want your feedback, we want to adapt material to your feedback
- Please feel free to contact us via email to ask questions at any time
- We'd love to reach out offline, discuss what's working, what to spend more time on
- For some orgs, may make sense for us to do a few bespoke sessions

Blog:
https://www.nag.com/blog/algorithmic-differentiation-masterclass-series-2

Outcomes

- Understand the idea of the vector versions for both AD models
  - tangent-linear and
  - adjoint model
- Learn how to compute the Jacobian using tangent-linear and adjoint vector model with `dco/c++`
- Learn how vector models in conjunction with AVX/SIMD instruction can speed up the computation of the Jacobian
- Continue discussion on which AD model should be used

Algorithmic Differentiation: Tangent-Linear Model

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

- **Tangent-Linear (scalar) Model (TLM)** $\dot{F}$

  $$\dot{F}(\boldsymbol{x}, \dot{\boldsymbol{x}}) = \underbrace{F'(\boldsymbol{x})}_{\in \mathrm{R}^{m \times n}} \cdot \underbrace{\dot{\boldsymbol{x}}}_{\in \mathrm{R}^n} = \dot{\boldsymbol{y}}$$

  - □ $\frac{Cost(\dot{F})}{Cost(F)} \approx 2$

- **Tangent-Linear (vector) Model (TLM)** $\dot{F}_k$

  $$\dot{F}_k(\boldsymbol{x}, \dot{\boldsymbol{X}}) = \underbrace{F'(\boldsymbol{x})}_{\in \mathrm{R}^{m \times n}} \cdot \underbrace{\dot{\boldsymbol{X}}}_{\in \mathrm{R}^{n \times k}} = \left(F'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}_1, \ldots, F'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}_k\right) = \underbrace{\dot{\boldsymbol{Y}}}_{\in \mathrm{R}^{m \times k}}$$

  - □ $\frac{Cost(\dot{F}_k)}{Cost(F)} \approx k + 1$

Jacobian with tangent-linear vector model

- In tangent linear **scalar** model we compute the Jacobian column-wise by setting $\dot{\boldsymbol{x}}$ to Cartesian basis vectors of $\mathbb{R}^n$ $e_i : 1 \leq i \leq n$. Especially

$$\dot{F}(\boldsymbol{x}, e_i) = F'(x) \cdot e_i = F'(x)_{-,i},$$

  where $F'(x)_{-,i}$ denotes the i-th column of $F'(x)$.

- In tangent linear **vector** model we compute k columns of the Jacobian simultaneously by setting $\dot{\boldsymbol{X}} = I_n^{i,k}$, where $I_n^{i,k} = (e_i, e_{i+1}, \ldots, e_{i+k})$ (an $n \times k$ block identity matrix). Hence

$$\dot{F}_k(\boldsymbol{x}, I_n^{i,k}) = F'(x) \cdot I_n^{i,k} = F'(x)(e_i, \ldots, e_{i+k}) = F'(x)_{-,i\ldots i+k}$$

  where $F'(x)_{-,i\ldots i+k}$ denotes the i-th to $i+k$-th column of $F'(x)$.

Inside tangent-linear vector model

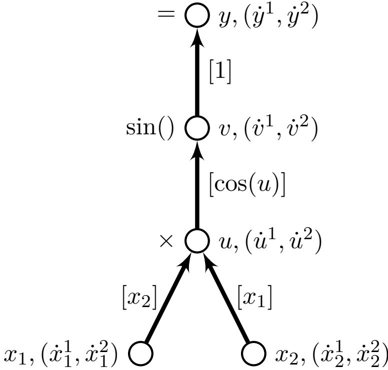
$$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$$

Tangent linear code (2 directions)

$\dot{u}^1 = x_2 \cdot \dot{x}_1^1 + x_1 \cdot \dot{x}_2^1$
$\dot{u}^2 = x_2 \cdot \dot{x}_1^2 + x_1 \cdot \dot{x}_2^2$
$u = x_1 \cdot x_2$

$\dot{v}^1 = \cos(u) \cdot \dot{u}^1 \qquad \dot{v}^2 = \cos(u) \cdot \dot{u}^2$
$v = \sin(u)$

$\dot{y}^1 = \dot{v}^1 \qquad \dot{y}^2 = \dot{v}^2$
$y = v$



Algorithmic Differentiation: Adjoint Model

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

- **Adjoint Model (scalar) (ADM)** $\bar{F}$ **(reverse mode)**

$$\bar{F}(\boldsymbol{x}, \bar{\boldsymbol{y}}) = \underset{\in \mathbf{R}^m}{\bar{\boldsymbol{y}}} \cdot \underset{\in \mathbf{R}^{m \times n}}{F'(\boldsymbol{x})} = F'(\boldsymbol{x})^T \cdot \bar{y} = \bar{\boldsymbol{x}}$$

  - $\frac{Cost(\bar{F})}{Cost(F)} = M$

- **Adjoint (vector) Model (ADM)** $\bar{F}_k$

$$\bar{F}_k(\boldsymbol{x}, \bar{\boldsymbol{Y}}) = \underset{\in \mathbf{R}^{n \times m}}{F'(\boldsymbol{x})^T} \cdot \underset{\in \mathbf{R}^{m \times k}}{\bar{\boldsymbol{Y}}} = \left(F'(\boldsymbol{x})^T \cdot \bar{\boldsymbol{y}}_1, \ldots, F'(\boldsymbol{x})^T \cdot \bar{\boldsymbol{y}}_k\right) = \bar{\boldsymbol{X}}$$

  - $\frac{Cost(\bar{F}_k)}{Cost(F)} = O(k) < M \cdot (k)$

Jacobian with adjoint vector model

- In adjoint **scalar** model we compute the Jacobian row-wise by setting $\bar{\boldsymbol{y}}$ to Cartesian basis vectors of $\mathbb{R}^m$ $e_i : 1 \leq i \leq m$. Especially

$$\bar{F}(\boldsymbol{x}, e_i) = F'(x)^T \cdot e_i = F'(x)_{i,-},$$

where $F'(x)_{i,-}$ denotes the i-th row of $F'(x)$.

- In adjoint **vector** model we compute k rows of the Jacobian simultaneously by setting $\bar{\boldsymbol{Y}} = I_m^{i,k}$, where $I_m^{i,k} = (e_i, e_{i+1}, \ldots, e_{i+k})$ (an $m \times k$ block identity matrix). Hence

$$\bar{F}_k(\boldsymbol{x}, I_n^{i,k}) = F'(x)^T \cdot I_n^{i,k} = F'(x)^T(e_i, \ldots, e_{i+k}) = F'(x)_{i \ldots i+k,-}$$

where $F'(x)_{i \ldots i+k,-}$ denotes the i-th to $i+k$-th row of $F'(x)$.

nag®

Inside adjoint vector model

$$F : \mathbb{R}^2 \to \mathbb{R}, \quad y = \sin(x_1 \cdot x_2)$$

Adjoint Code (2 directions)

Forward Sweep (primal computation)

$u = x_1 \cdot x_2$

$v = \sin(u)$

$y = v$

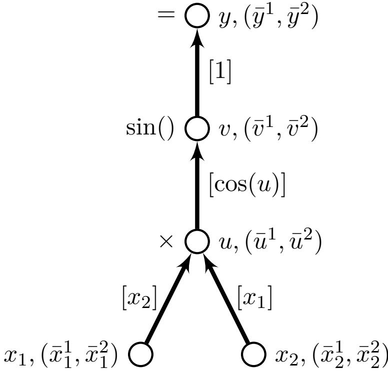
Reverse Sweep (adjoint computation)

$\bar{v}^1 = \bar{y}^1 \qquad \bar{v}^2 = \bar{y}^2$

$\bar{u}^1 = \cos(u) \cdot \bar{v}^1 \qquad \bar{u}^2 = \cos(u) \cdot \bar{v}^2$

$\bar{x}_1^1 = x_2 \cdot \bar{u}^1 \qquad \bar{x}_1^2 = x_2 \cdot \bar{u}^2$

$\bar{x}_2^1 = x_1 \cdot \bar{u}^1 \qquad \bar{x}_2^1 = x_1 \cdot \bar{u}^1$



Tangent-linear vector Model with dco/c++

- Replace floating point variables with `dco::gt1v<double, k>::type`, where `k` is the size of the tangent vector
- Write the driver
- Conceptually `dco::gt1v<double, k>::type` contains two components, both components are computed during normal execution
  - value
  - tangent (array)
- Interface
  - `dco::value(DCO_TYPE)` - access to value component
  - `dco::derivative(DCO_TYPE)` - access to the tangent (array) component

Tangent-Linear vector Model: Jacobian with dco/c++

```
template <typename T>
void foo(int& n, T* x, T& y){ ... }
int main(){
  DCO_TANGENT_VECTOR_TYPE *x, y;
  ...
  for (int i = 0; i < n; i+=k) {
    for (int j = 0; j < k;j++)
      dco::derivative(x[i+j])[j] = 1.0;

    foo(n, x, y);

    for (int j = 0; j < k; j++)
      J[i+j] = dco::derivative(y)[j];
    for (int j = 0; j < k; j++)
      dco::derivative(x[i+j])[j] = 0.0;
}
```

The tangent-linear vector model of `foo` is executed n/k times

Tangent-linear vector Model

Why should I use it?

- each time tangent-linear (scalar/vector) model is called the function value is computed along with tangent-linear projections.
- with tangent-linear vector model we perform only n/k primal function evaluations instead of n with tangent-linear scalar model
- computing several projections at the same time can profit from AVX/SIMD extensions on your CPU.

Problems

- higher memory requirements to store the additional tangents
- more complicated driver

Tangent-Linear vector Model

How should I choose the vector size for `gt1v`?

- Choosing the vector size as big as possible (`k=n`)
  - reduces the number of primal function evaluations
  - high memory requirements. Each `gt1v` variable contains $n + 1$ doubles
  - bad memory access pattern (caching effects)

- On modern Intel CPU's with AVX2/AVX512 vector sizes (8/12/16) is almost always the best choice.
  - choose vector sizes as multiple of four/eight to be aligned with AVX2/AVX512
  - the unused directions should be set to zero (number of inputs not multiple of the chosen vector size).

Computing several adjoint directions with `ga1s`

```
void foo(const int &n, const int &m, T* x, T* y) {...}
int main(){
  DCO_ADJOINT_VECTOR_TYPE *x, *y;
  ...
  foo(n, m, x, y); //Record the tape
  for (int i = 0; i < m; i++) { //m tape interpretations
    dco::derivative(y[i]) = 1.0; //set adjoint of y

    DCO_M::global_tape->interpret_adjoint();

    for (int j = 0; j < n; j++) //Extract the derivatives
      J[i*n + j] = dco::derivative(x[j]);
    //zero all adjoints
    DCO_M::global_tape->zero_adjoints();
  }
}
```

Function `foo` is recorded once but is interpreted m times

Computing several adjoint directions: Remarks

- Record the function only once and interpret as many times as you need
- Interpretation step is much cheaper compared to recording step. Only 10%-30% of the overall time.
  - during recording the DAG (tape) is written to memory including computation of partial derivatives
  - tape interpretation performs only fma's operations. DAG (tape) is left untouched only the adjoint vector is updated.
- Vector adjoint model - can save only the interpretation time. Recording time is the same.

  NOTE: All information only applies to adjoint model not using advanced adjoint techniques (e.g. checkpointing, symbolic adjoints)

Adjoint vector Model with `dco/c++`

- Replace floating point variables with `dco::ga1v<double, k>::type`, where `k` is the number of the adjoint directions computed simultaniously
- Write the driver

- Conceptually `dco::ga1v<double, k>::type` contains two components
  - value
  - adjoint (array)

  During the execution of the function `dco/c++` computes the value component and records the computational graph (tape). Interpretation of the tape is needed to compute the adjoint components.

Adjoint vector Model with dco/c++: Basic Interface

- Interface of `dco::ga1v<double, k>::type`
  - `dco::value(DCO_TYPE)` - access to value component
  - `dco::derivative(DCO_TYPE)` - access to the adjoint (array) component
- Interface of the tape `DCO_MODE::tape_t`
  - `DCO_TAPE_TYPE::create()` - creates tape and returns pointer to it
  - `DCO_TAPE_TYPE::remove(DCO_TAPE_TYPE*)` - deallocates tape
  - `DCO_TAPE_TYPE::register_variable(DCO_TYPE)` - Marks variable as independent
  - `DCO_TAPE_TYPE::register_output_variable(DCO_TYPE)` - Marks variable as dependent
  - `DCO_TAPE_TYPE::interpret_adjoint()` - Runs tape interpreter

Adjoint vector Model with dco/c++: Jacobian

```
template <typename T>
void foo(int& n, int& m, T* x, T& y){ ... }
int main(){
  DCO_ADJOINT_VECTOR_TYPE *x, y;
  ...
  for (int i = 0; i < m; i+=k) {
    for (int j = 0; j < k; j++)
     dco::derivative(y[i+j])[j] = 1.0;
   DCO_M::global_tape->interpret_adjoint(); //Interpret
   for (int s = 0; s < k; s++) {
     for (int j = 0; j < n; j++)
       J[(i+s) * n + j] = dco::derivative(x[j])[s];
   }
   DCO_M::global_tape->zero_adjoints();
  }
}
```

Function `foo` is recorded once but is interpreted m/k times

Adjoint vector Model

Why should I use it?

- with adjoint vector model we perform only m/k tape interpretations instead of m with adjoint scalar model. Can improve performance when used in conjunction with advanced adjoint techniques (checkpointing, symbolic adjoints)
- computing several adjoint projections at the same time can profit from AVX/SIMD extensions on your CPU.

Problems

- higher memory requirements to store the additional adjoints directions. Can be addressed by using compressed adjoint vector (Reuse adjoint storage by analysing the maximum number of required distinct adjoint memory locations (Master Class 5)).
- more complicated driver

Adjoint vector Model

How should I choose the vector size for `ga1v`?

- Choosing the vector size as big as possible (`k=m`)
  - reduces the number of tape interpretations
  - high memory requirements (Each entry in the vector of adjoints is now a vector of doubles with size `k`.)
  - bad memory access pattern (caching effects)

- On modern Intel CPU's with AVX2/AVX512 vector sizes (4/8/12/16) is almost always the best choice.
  - choose vector sizes as multiple of four/eight to be aligned with AVX2/AVX512
  - the unused directions should be set to zero (number of outputs not multiple of the chosen vector size).

Tangents vs Adjoints

In Masterclass 2 we already learned that if $M \cdot m < 2 \cdot n$ you should use the adjoint model and else the tangent-linear model. Now let's see how the tangent vector model can change this statement:

Pro tangent

- Tangent vector mode can speed up your tangent computation by a factor of ≈ 3 on a AVX2 machine. AVX512 should provide even better results.
- Hence for $m = 1$ we should rather say that $M < n$ to justify the usage of adjoints
- For $m > 1$ please note that adjoint model can be very efficient on computing several rows of Jacobian (see next slide).

Tangents vs Adjoints

In Masterclass 2 we already learned that if $M \cdot m < 2 \cdot n$ you should use the adjoint model and else the tangent-linear model. Now let's see how the adjoint vector model can change this statement:

- **Pro adjoint**
  - ☐ Computing additional directions with adjoint **scalar** model is $\approx 10\% - 30\%$ overall runtime. (without advanced adjoint techniques)
  - ☐ Using adjoint vector mode provides an addition speed up of ≈ 3 on a AVX2 machine compared to the adjoint scalar model. AVX512 machine should perform even better.
  - ☐ Even for big m adjoint might be the right choice, if the tape is small enough. Especially if $M < 10$.

Tangents vs Adjoints

- Consider using all available models and know their strength and weaknesses. Don't do adjoint because somebody tells you it is great.
- Start with the implementation of the tangent model (you need it anyway as discussed in Master Class 3)
- Based on performance on tangent-linear model you can decide, whether you need the adjoint model
  - $M > 30$ - naive implementation of adjoints should give you desired performance
  - $M < 20$ - you will need some tweaks (e.g. symbolic adjoints)
  - $M < 10$ - very good adjoint code significant development time required
- Adjoint requires control flow reversal, making it challenging for parallel codes and increases the memory usage (forcing to use advanced adjoint techniques).

Summary

In this Class we learned

- How tangent-linear and adjoint vector model work.
- How they can be used to speed up the computation
- How AVX/SIMD extension can help us speed up
  - tangent-linear vector Model
  - adjoint vector Model
- Continue discussion on tangent vs. adjoints

Bootstrapping validated adjoints on real-world codes

In the next part our we will learn different automatic techniques to avoid running out of memory when computing the adjoints such as

- write tape to disk
- Jacobian pre-accumulation
- compressing the vector of adjoints

The goal to compute an adjoint result that we can use as a reference for potential later code optimisations.

You will see a survey on your screen after exiting from this session.

We would appreciate your feedback.

We are now moving on the Q&A Session