

GPU Trajectory Optimization

with some discussion of “small” linear algebra on the GPU

Josh Bialkowski

jbialk@mit.edu

December 16, 2012

1 Introduction

The goal of this project was to investigate the runtime savings of implementing trajectory optimization for a complex rigid body system using direct collocation on a GPU. The NLP transcription of such a system is a significant undertaking in itself. For a second order rigid body system in three dimensions the size of the state is at least 12 dimensions. Thus the NLP transcription requires a derivation of the first and second order sensitivities for at least 12 decision variables, resulting in $12 + 12^2/2$ equations. While this full system is the desired end goal, for this project I implemented trajectory optimization for a much simpler model in Nvidia CUDA C++ and compared it’s runtime to that of an equivalent implementation in serial C++ on a single processor.

In addition to the large number of equations required for the real problem, there is also the issue of reliably implementing them in code. In particular, many of the equations may be compactly represented using linear algebra. There are great C++ libraries for linear algebra on the CPU with adequate operator overloads to map the usual arithmetic operators to their linear algebra counterparts (a personal favorite is the **Eigen** library). However there is a notable absence of such libraries for performing (small) matrix/vector computations within a GPU kernel (alas **Eigen** cannot be compiled for CUDA device code). Having a reliable and efficient library for these operations will allow a more direct transcription of the sensitivity equations from a mathematical representation to C++ code and reduce the likelihood of transcription error in this process (and with so many equations to transcribe, that likelihood is quite large). Thus, I also investigate some methods of adapting standard C++ techniques from modern linear algebra libraries to CUDA, and analyze the performance of my baseline linear algebra library in a matrix reduction kernel.

2 Trajectory optimization

The trajectory optimization problem is essentially the problem of finding a state and control trajectory which optimizes some cost functional and satisfies the dynamic constraints of the system. Formally:

$$\begin{aligned} & \underset{\mathbf{x}(t), \mathbf{u}(t), t_f}{\text{minimize}} && J(\mathbf{x}(t), \mathbf{u}(t), t) \\ & \text{subject to} && \dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t) \\ & && \mathbf{x}(0) = \mathbf{x}_0 \\ & && \mathbf{x}(t_f) = \mathbf{x}_f \\ & && \mathbf{x} : [0, t_f] \rightarrow X \\ & && \mathbf{u} : [0, t_f] \rightarrow \mathcal{U} \end{aligned} \tag{1}$$

We generally require the state trajectory \mathbf{x} to be continuous and we may also require that the state trajectory and control inputs \mathbf{u} to satisfy continuity of one or more derivatives.

One method of approximately solving the trajectory optimization problem is to transcribe the continuous optimization into a discrete nonlinear program (NLP). In the method of direct collocation we assume that the state and control trajectories can be described by some equation with a finite parametrization (for instance, a polynomial spline). Then we use standard NLP solvers to find the set of parameters which optimize the cost functional.

In particular I chose to follow the method of [2]. In this method we discretize the path into N intervals with the j th interval covering Δt_j time. We choose as decision variables $\mathbf{x}_j, \mathbf{u}_j$ the state and control input at the junction of each interval. Then we describe the interval by a polynomial spline incorporating the desired continuity constraints. For instance, we may guarantee second order continuity by a fifth order polynomial. Thus for a particular state variable x on the j th interval we assume it follows the form

$$\begin{aligned} x_j(t) &= c_0 + c_1 s + c_2 s^2 + c_3 s^3 + c_4 s^4 + c_5 s^5 \\ s(t) &= \frac{t - t_j}{\Delta t_j} \end{aligned} \quad (2)$$

We may compactly represent the polynomial as

$$\begin{aligned} x(s(t)) &= \mathbf{c}^\top \mathbf{d}_0(s) \\ \mathbf{c} &= [c_0 \quad c_1 \quad c_2 \quad c_3 \quad c_4 \quad c_5]^\top \\ \mathbf{d}_0(s) &= [1 \quad s \quad s^2 \quad s^3 \quad s^4 \quad s^5]^\top \end{aligned} \quad (3)$$

Then we can compactly represent its first and second derivative with respect to time

$$\begin{aligned} \dot{x}(s) &= \mathbf{c}^\top \mathbf{d}_1(s) \\ \ddot{x}(s) &= \mathbf{c}^\top \mathbf{d}_2(s) \\ \mathbf{d}_1(s) &= \frac{1}{\Delta t_j} [0 \quad 1 \quad 2s \quad 3s^2 \quad 4s^3 \quad 5s^4]^\top \\ \mathbf{d}_2(s) &= \frac{1}{\Delta t_j^2} [0 \quad 0 \quad 2 \quad 6s \quad 12s^2 \quad 20s^3]^\top \end{aligned} \quad (4)$$

Then, in order to guarantee second order continuity at the knots between the j th interval and it's neighbors we have the following equations:

$$\begin{aligned} x(0) &= x_j & x(1) &= x_{j+1} \\ \dot{x}(0) &= \dot{x}_j & \dot{x}(1) &= \dot{x}_{j+1} \\ \ddot{x}(0) &= \ddot{x}_j & \ddot{x}(1) &= \ddot{x}_{j+1} \end{aligned} \quad (5)$$

Which means the coefficients must satisfy

$$\begin{aligned} \begin{bmatrix} \mathbf{d}_0(0)^\top \\ \mathbf{d}_1(0)^\top \\ \mathbf{d}_2(0)^\top \\ \mathbf{d}_0(1)^\top \\ \mathbf{d}_1(1)^\top \\ \mathbf{d}_2(1)^\top \end{bmatrix} \mathbf{c} &= \begin{bmatrix} x_j \\ \dot{x}_j \\ \ddot{x}_j \\ x_{j+1} \\ \dot{x}_{j+1} \\ \ddot{x}_{j+1} \end{bmatrix} \\ D\mathbf{c} &= \mathbf{y}_j \end{aligned} \quad (6)$$

Therefore we may simply describe the state and it's derivatives along the j th interval by a function which

is polynomial in the knot values:

$$\begin{aligned} x^{(k)}(s) &= (D^{-1}\mathbf{y}_j)^\top \mathbf{d}_k(s) \\ x^{(k)}(s) &= \mathbf{y}_j^\top D^{-\top} \mathbf{d}_k(s) \\ x^{(k)}(s) &= \mathbf{y}_j^\top \mathbf{b}_k(s) \end{aligned} \tag{7}$$

The cost then is the accumulated cost on each interval: $J = \sum_{j=0}^N J_j$. We incorporate the constraints by forming the Legrangian

$$\begin{aligned} L &= J + \boldsymbol{\lambda}^\top \mathbf{F} \\ \mathbf{F} &= \dot{\mathbf{x}} - f(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{8}$$

We may choose to enforce the constraints only at specific points in order to form a discrete NLP. However, given the parametrization for the states on each interval, we may enforce the constraints at the knots directly. Then we choose as the constraints for the NLP the value of $\mathbf{F}_j(s)$ at some point s along the j th interval. Thus the Legrangian for the NLP is given by

$$L = \sum_{j=0}^N J_j + \boldsymbol{\lambda}_j^\top \mathbf{F}_j(s) \tag{9}$$

The optimal cost satisfying the constraints is found at a saddle point of the Legrangian. We may either search for the saddle point or choose to minimize the square of the Legrangian $J = L^2$. In this case we may use a descent method such as Newton's method to find the optimal decision variables. Let \mathbf{X}_j be the row concatenation of the decision variables \mathbf{x}_j , \mathbf{u}_j , and $\boldsymbol{\lambda}_j$, and let \mathbf{X} be the row concatenation of \mathbf{X}_j for $j = 1 \dots N$. Then at each iteration we choose the step direction $\Delta \mathbf{X}$ to minimize the cost by a second order approximation:

$$\begin{aligned} J_{i+1} &\approx J_i + \nabla J(\Delta \mathbf{X}) + \frac{1}{2}(\Delta \mathbf{X})^\top \nabla^2 J(\Delta \mathbf{X}) \\ \Delta \mathbf{X} &= (\nabla^2 J)^{-1} \nabla J \end{aligned} \tag{10}$$

3 System of Interest

I am particularly interested in the problem of optimal trajectory planning for a small unmanned propeller driven fixed wing aircraft through a cluttered environment (i.e. a forest). We can derive a simple aerodynamic model based on flat-plate aerodynamic theory if we assume that the body of the aircraft can be roughly described by a pair of orthogonal flat sections, and that the control surfaces do not contribute significantly to a change in the net aerodynamic force on the vehicle. The system evolves on the group $SE(3)$ and the equations of motion for the system are given by the Newton-Euler equations for rigid body dynamics, with the aerodynamic model for the forces.

$$\begin{aligned} m\dot{\mathbf{v}} &= -\boldsymbol{\omega} \times m\mathbf{v} + \mathbf{F}_b + R\mathbf{g} \\ J\dot{\boldsymbol{\omega}} &= -\boldsymbol{\omega} \times J\boldsymbol{\omega} + \mathbf{M}_b \end{aligned} \tag{11}$$

Where the inertia of the system are the mass m and moment of inertia J , \mathbf{v} is the velocity in the body frame, $\boldsymbol{\omega}$ is angular velocity in the body frame, R is a rotation matrix which rotates a vector in global coordinates to a vector in the body frame, \mathbf{g} is the gravitational force vector, and \mathbf{F}_b and \mathbf{M}_b are the body forces and moments.

From the simple aerodynamic force model the body forces are given by:

$$\mathbf{F}_b = \begin{bmatrix} T \\ -kh\sqrt{v_0^2 + v_2^2}v_2 \\ -kv\sqrt{v_0^2 + v_1^2}v_1 \end{bmatrix} \tag{12}$$

where T is the thrust, and v_i is the i th component of \mathbf{v} . The equations of motion are given in the body frame, but as we are concerned with the path the vehicle takes, we will perform the trajectory optimization in a global (inertial) frame. In the inertial frame we will describe the state of the system as:

- position (of the center of mass): \mathbf{p}
- it's derivatives: $\dot{\mathbf{p}}, \ddot{\mathbf{p}}$
- orientation: \mathbf{r} , a rotation vector describing the transformation of a vector in global coordinates to the equivalent vector represented in body coordinates
- it's derivatives: $\dot{\mathbf{r}}, \ddot{\mathbf{r}}$
- the thrust input T

This model is tremendously complex, so as a first step towards realizing an implementation of this trajectory optimization, let us consider the Dubins car model of an aircraft in the plane. The Dubins car model assumes that the aircraft is flying with a fixed forward speed, and has a limited turning radius:

$$\begin{aligned}\dot{p}_x &= v \cos \theta \\ \dot{p}_y &= v \sin \theta \\ \dot{\theta} &= u\end{aligned}\tag{13}$$

Note that the shortest path for a dubins car between two given states has an analytical solution, but provides a useful test case for implementing trajectory optimization. We may approximate the path cost by the distance between consecutive knots:

$$J_j = \sqrt{(p_{x_{j+1}} - p_{x_j})^2 + (p_{y_{j+1}} - p_{y_j})^2}\tag{14}$$

Since this is only a first order system we will describe the position trajectory, $\mathbf{p}(t)$ by a cubic spline, and the orientation trajectory $\theta(t)$ as piecewise linear. For the sake of brevity, I will forgo reporting the derivation of the cost sensitivities in this report.

4 Profiling Results

I implemented a descent solver for this system in C++ as a CUDA kernel and a serial version for comparison. The code may be found at [git://git.cheshirekow.com/mpblocks/examples/cuda_trajopt.git](https://git.cheshirekow.com/mpblocks/examples/cuda_trajopt.git). The implementation assigns a single trajectory optimization (one start/end state pair) to a single CUDA *block* (essentially, a single physical processor). The GPU kernel has a maximum block allocation of 380 threads. I profiled the runtime of performing the Dubins trajectory optimization in serial and on the GPU on two separate machines. A laptop with an Intel(R) Core(TM) i7 Q820 CPU at 1.73GHz and a Nvidia Quadro MX 1800 GPU, and a server (ares) with an Intel(R) Xeon(R) E5620 CPU at 2.40GHz and an Nvidia Tesla C2075 GPU. The results for a single start/end state pair and for a batch of ten start/end state pairs are shown in Figures 1 and 2.

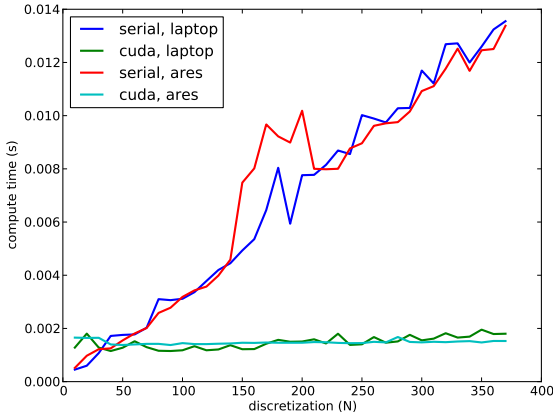


Figure 1: runtime for a single start/end state pair

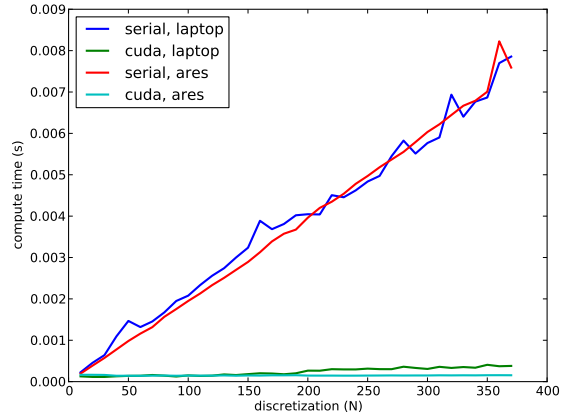


Figure 2: average runtime for 10 start/end state pairs

The runtime results show a dramatic improvement in compute time, which is further realized by batching the multiple start/end state pairs. I also profiled the runtime for a fixed 200 discretization intervals, and multiple batch sizes. The results are shown in Figure 3.

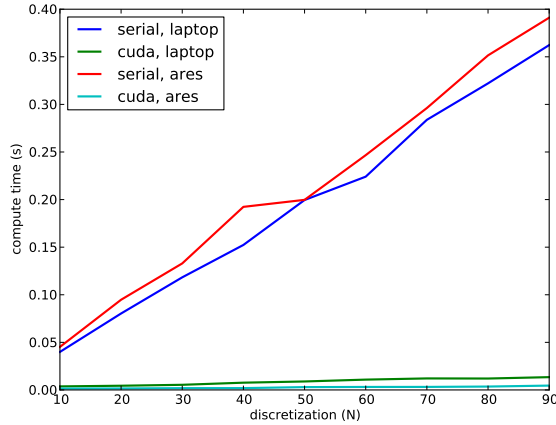


Figure 3: runtime for multiple batch sizes and 200 discretization points

While the improvement is significant for the Dubins car model, I will note that the model is sufficiently small in size that a single thread is capable of storing the state pair and sensitivities for an entire interval. This simplified implementation allows for a straightforward one-thread-per-interval decomposition of the problem. For the full system, I suspect that the number of values that need to be calculate will force me to move the computation of values for a single interval across several threads. This will reduce the number of intervals that can be computed on a single physical processor, and increase the amount of shared memory for communication across threads. Since the number of intervals that may fit on a single processor will go down, this may mean that I will have to change how the problem is broken down, and resort to communication across physical processors.

5 Linear algebra on the GPU

As previously mentioned, one barrier to implementing trajectory optimization for larger systems is the lack of a quality linear algebra library for performing matrix/vector calculations *within* threads. There has been a lot of focus on performing large scale linear algebra on the GPU, and there are libraries for doing exactly this. The Nvidia CUDA libraries include an implementation of BLAS for use in BLAS based codes. However I was unable to find an implementation of a linear algebra library for performing small scale operations within a single thread. In order to overcome this hurdle I have attempted to write such a library that provides standard C++ operator overloads and linear algebra routines providing the ability to perform basic vector/matrix math inside a GPU kernel.

The limitations of GPU hardware make this quite difficult as there are many things to consider. For one: there is a very limited amount of high speed memory (registers) available to each thread, and that memory is easily saturated by the creation of temporary objects. When this happens, the kernel experiences what is denoted in the CUDA documentation as *register spilling*. When the compiler runs out of registers to use in the kernel, it starts storing variables in *local* memory. Local memory, however, is really just a thread-private address space within *global memory* and GPUs have very slow access to global memory (memory accesses are high latency). While some of the latest cards have L1 and L2 cache, most of the cards in use today do not. Thus, it is important to make efforts to keep as much of the memory as possible within thread registers.

In the context of linear algebra, this means there is a lot to be gained by eliminating the need for temporary objects. One strategy for such optimizations within the C++ community is defer computation until it is needed by use of *template expressions*. Consider, for example, the following equation:

$$c = s(Ax + b)$$

where A , is a matrix, c , x , and b are vectors, and s is a scalar. In a naive implementation of a linear algebra library we may define the product (*) and addition (+) operators with something similar to the following signatures:

```
Matrix operator*( const Matrix& A, const Matrix& B );
Matrix operator+( const Matrix& A, const Matrix& B );
```

The problem with this is that we have the somewhat unnecessary creation of a temporary object for each of these operations (with the caveat that the compiler may often optimize out these temporaries). Thus $A*x$ creates a temporary vector, it's sum with b creates a temporary, and then finally the product with the scalar s creates a third. The main idea behind template expressions is that the binary operators do not return the result of the operation, but returns an object which simply wraps a reference to the operands, and provides an implementation for access operators which performs the computation when that element is requested. More over, these implementations are usually inlined and provide an avenue for significant improvement by compiler optimizations. As an example, here is an excerpt of the implementation of the sum template expression in my library (found in `include/mpblocks/cuda/linalg/Sum.h` in the git repository).

```
/// expression template for sum of two matrix expressions
template <typename Scalar, class Exp1, class Exp2>
class Sum :
    public RValue<Scalar, Sum<Scalar,Exp1,Exp2> >
{
    Exp1 const& m_A;
    Exp2 const& m_B;

public:
    /// return the evaluated i'th element of a vector expression
    Scalar operator[]( Size_t i ) const
```

```

    {
        return (m_A[i] + m_B[i]);
    }

    /// return the evaluated (i,j)'th element of a matrix expression
    Scalar operator()( Size_t i, Size_t j )const
    {
        return ( m_A(i,j) + m_B(i,j) );
    }
};

template <typename Scalar, class Exp1, class Exp2>
Sum<Scalar,Exp1,Exp2> operator+(
    RValue<Scalar,Exp1> const& A, RValue<Scalar,Exp2> const& B )
{
    typedef Sum<Scalar,Exp1,Exp2> Sum_t;
    return Sum_t(
        static_cast<Exp1 const&>(A),
        static_cast<Exp2 const&>(B));
}

```

Note that the addition operator (+) takes in two `RValue` references, and it returns nothing more than an object which references these (i.e. the temporary object uses at most the size of two pointers in memory). The `Sum` object is also an `RValue` so it can be used by any other template expressions. The net results is that when the access operator `[i]` or `(i,j)`, are used, the program navigates down the tree of expression objects and builds the result to be returned.

In my linear algebra library I have implemented template expressions for matrix/vector addition, subtraction, multiplication, multiplication by a scalar, transpose, views, block assignment, and initialization by streams. These may be found in `include/mpblocks/cuda/linalg` in the git repository. In order to evaluate the performance of these routines, I implemented a matrix reduction kernel, which computes the product of many small matrices (similar to addition reduction kernel provided in the CUDA SDK examples). I followed the guidelines of the Nvidia developer presentation regarding optimizing reduction kernels, found at http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. I implemented four different kernels attempting to address some of the issues with optimizing the reductions. The runtime for computing the product of $2^{13} 3 \times 3$ matrices for two different machines is shown in Tables 1 and 2 under the column “dynamic”. These runtimes are averaged over 30 consecutive calls to the kernel in order to average out variances in the system overhead, and do not include the time to transfer the data to and results from the GPU to the host.

Kernel 0 is a naive first implementation. Kernel 1 attempts to synchronize thread accesses to local memory. Kernel 2 attempts to reduce bank conflicts for threads accessing shared memory (which isn’t as successful since the matrices are not aligned to the shared memory pitch), and Kernel 3 unrolls the last 8 loops of the reduction (since these occur in the same warp and are synchronized by hardware). We see a rather significant speedup of Kernel 3 over Kernel 0 (about 2x) but the difference between Kernels 1 and 2 are minimal. On the laptop graphics card they seem to be approximately the same as Kernel 0, but there is an improvement on the server graphics card.

| kernel | dynamic | static |
|--------|---------|--------|
| 0 | 1.0761 | 0.9593 |
| 1 | 0.9855 | 0.8560 |
| 2 | 1.1999 | 0.1513 |
| 3 | 0.4162 | 0.1477 |

Table 1: Average runtime in ms over 30 kernel calls on a Quadro MX 1800M

| kernel | dynamic | static |
|--------|---------|--------|
| 0 | 0.2824 | 0.1656 |
| 1 | 0.1642 | 0.1599 |
| 2 | 0.1681 | 0.1458 |
| 3 | 0.1807 | 0.1481 |

Table 2: Average runtime in ms over 30 kernel calls on a Tesla C2075

One reason for the difference between the two graphics cards may be the result of register spilling. If we look at register and local memory usage for these kernels on the two GPUs we see the following:

| k | impl 1 | | impl 2 | |
|---|--------|-------|--------|-------|
| | Quadro | Tesla | Quadro | Tesla |
| 0 | 20 | 28 | 28 | 29 |
| 1 | 20 | 28 | 21 | 26 |
| 2 | 14 | 26 | 15 | 29 |
| 3 | 21 | 27 | 24 | 30 |

Table 3: Register usage (number of 32 bit registers) for matMul kernel

| k | impl 1 | | impl 2 | |
|---|--------|-------|--------|-------|
| | Quadro | Tesla | Quadro | Tesla |
| 0 | 72 | 0 | 0 | 0 |
| 1 | 36 | 0 | 0 | 0 |
| 2 | 72 | 0 | 0 | 0 |
| 3 | 288 | 0 | 0 | 0 |

Table 4: Local memory usage (in bytes) for matMul kernel

On the laptop GPU (which is of lesser compute capability) we’re using a rather significant amount of local memory. Local memory is in fact just global memory private to each thread. Global memory has rather high latency so it’s a good idea to try to avoid it. There are two reasons for register spilling in a GPU kernel. One is simply running out of registers. If the GPU kernel requires more than a certain maximum number of registers for storage of thread-local variables, the compiler will allocate local memory for these variables. The other reason for register spilling is dynamically addressing addressing in arrays. For instance if declare an array of floats like follows:

```
float arr[10];
```

this may indicate that ten 32 bit registers are reserved for floating point numbers. However, if we attempt to access these values in the following way

```
for(int i=0; i < 10; i++)
    arr[i] = 10*i;
```

we are attempting to address these values via pointer + offset arithmetic. Because the location of the access may not be known at compile time, the compiler may move the array `arr` into local memory. I suspect this is the cause of register spilling in the GPU kernels on the laptop card. It’s curious that the server card isn’t using local memory for the same kernels. That machine, however, has a newer version of the CUDA toolkit so it’s possible that the updated compiler is able to optimize around this dynamic indexing. In any case, let’s assume that this addressing is in fact slowing us down.

The CUDA compiler implements most of the C++ template facilities, so we may look for a way of providing a library of *statically sized* matrix objects, with methods for *statically* indexing them. Thus we can provide a usual interface for performing linear algebra operations, without forcing the compiler to move the objects into local memory, or having to use dynamic addressing for accessing members.

I’ve attempted to accomplish this by combining the methods of template expressions with template member functions where the indices of the accessors are in fact template parameters (known at compile

time) instead of function parameters. In this way the accessor for the (0,1)th element of a matrix A can be used like this:

```
A.me<0,1>() = 10;
```

The accessor `Matrix::me<0,1>(void)` is in fact a *separate* function from `Matrix::me<i,j>(void)` for any i, j . Furthermore, we can utilize template specialization and template self-inheritance to build a matrix class which subclasses one copy of an element storage class for each element it requires. An excerpt of this implementation is below (from `mpblocks/cuda/linalg2/Matrix.h`)

```
/// default template has column inheritance
template <typename Scalar, Size_t ROWS, Size_t COLS, Size_t i, Size_t j>
struct MatrixElement :
    public MatrixElement<Scalar, ROWS, COLS, i, j+1>
{
    Scalar m_data;
};

/// specialization for 0'th column, also inherits row
template <typename Scalar, Size_t ROWS, Size_t COLS, Size_t i>
struct MatrixElement<Scalar, ROWS, COLS, i, 0> :
    public MatrixElement<Scalar, ROWS, COLS, i, 1>,
    public MatrixElement<Scalar, ROWS, COLS, i+1, 0>
{
    Scalar m_data;
};

/// empty class for last row
template <typename Scalar, Size_t ROWS, Size_t COLS>
struct MatrixElement<Scalar, ROWS, COLS, ROWS, 0>
{};

/// empty class for last column
template <typename Scalar, Size_t ROWS, Size_t COLS, Size_t i>
struct MatrixElement<Scalar, ROWS, COLS, i, COLS>
{};

template <typename Scalar, Size_t ROWS, Size_t COLS>
class Matrix :
    protected MatrixElement<Scalar, ROWS, COLS, 0, 0>,
    public LValue< Scalar, ROWS, COLS, Matrix<Scalar, ROWS, COLS> >,
    public RValue< Scalar, ROWS, COLS, Matrix<Scalar, ROWS, COLS> >
{
public:
    typedef Matrix<Scalar, ROWS, COLS>          Matrix_t;
    typedef LValue<Scalar, ROWS, COLS, Matrix_t > LValue_t;

public:
    /// vector accessor
```

```

template <Size_t i>
__device__ __host__
Scalar& ve()
{
    return static_cast<
        MatrixElement<Scalar,ROWS,COLS,i,0>& >(*this).m_data;
}

/// matrix accessor
template <Size_t i, Size_t j>
__device__ __host__
Scalar& me()
{
    return static_cast<
        MatrixElement<Scalar,ROWS,COLS,i,j>& >(*this).m_data;
}
};

```

Because a `Matrix` is a `RValue` it can be used in the RHS of any template expressions. Because it is a `LValue` it accepts assignment from an `RValue` expression. By the tree of inheritance the template instantiation for `Matrix<float,3,3>` will inherit from nine separate instances of the `MatrixElement` class template. Thus it will have storage for nine `floats`, and the accessor function templates instantiate a separate function to access each element. Furthermore, because these are all POD types with no virtual inheritance, a `Matrix` is a POD data type and it can be cast directly to a C-array of `Scalars` for dynamic addressing (though the look-up depends on how the compiler orders the elements).

The trade-off we make is increased program size. We now have N^2 separate functions for each of our template expressions (as well as iterator expressions in calculating the product). I have implemented all the same expressions as in the original linear algebra library using compile-time addressing, as well as an appropriate cast-and-lookup for dynamic addressing using the layout produced by my compiler. The code for this library can be found in `include/mpblocks/cuda/linalg2` in the git repository. The runtime results for this library are also shown in Table 1 and 2 under the column “static”. We see that the static addressing does in fact lead to a significant improvement on the laptop GPU (Quadro MX). There is also some improvement in the server GPU, though not as significant. Table 4 shows that in this second implementation we have in fact eliminated any register spilling and the matrix reduction kernel now uses no local memory, even on the older laptop card.

6 Conclusion and future work

The implementation of the trajectory optimization for the Dubins car model leads me to believe that an implementation for the full aircraft model may realistically see significant gains over of serial implementation. My ultimate desire is to use trajectory optimization as a subroutine in the inner loop of a sampling-based planning algorithm, thus having a fast implementation for batched trajectory optimizations is crucial for the implementation of the full algorithm to have a reasonable runtime. I’m encouraged by the results and will move forward with my implementation of the full system.

In addition, I believe that the profiling results of my linear algebra library are quite satisfactory. The implementation provides much of the functionality I would look for and will allow me to create a more direct transcription from the complicated math that falls out of evaluating the sensitivities into correct-by-design C++ code.

References

- [1] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors, 2006.
- [2] Oskar von Stryk. Numerical solution of optimal control problems by direct collocation. In *in Optimal Control, (International Series in Numerical Mathematics 111)*, pages 129–143, 1993.