AMS 148 Chapter 7: N-Body Solver and Iterative Methods for Solving Linear Systems

Steven Reeves

Much of the material presented in this chapter comes from Chapter 31, of GPUGEMS3.

1 N-Body Simulation

An N-body simulation numerically approximates the evolution of a systme of bodies in which each body continuously interacts with every other body. An example of an N-body system is an astrophysical simulation in which each body represents a galaxy or an indiviual star, and the bodies interact through the gravitational force. N-body simulation arises in many other computational science problems such as:

- Protein folding to calculate the electrostatic and van der Waals forces
- Turbulent fluid flow simulation as particles
- Global illumination in computer graphics

1.1 Method

The *all-pairs* method to N-body calculations is a brute-force approach that computes all pair-wise interactions among the N bodies. It is a simple method, but not used on its own in large simulations as it is $\mathcal{O}(N^2)$. Generally, an all-pairs approach is used as a kernel to determine the forces in interactions of close range. This method is combined with a faster alogrithm based on a far-field approximation of longer-range forces – which is generally only valid for portions of the system that are separated. Faster N-body algorithms of this type include: particle-mesh method (Hockney and Eastwood 1981), the Barnes-Hut method (Barnes and Hut 1986), adn the fast multipole method (FMM, Greengard 1987).

The all pairs portion of the above algorithms contain the most compute instensive operation, and is therefore the best candidate for GPU acceleration. Accelerating the all-paris compoenent will improve the far-field calculations as well, since the balance between far-field and near body can be rebalanced to assign more work to a faster all-pairs component.

We will focus on the all-pairs computational kernel and its implementeation using CUDA. Parallelism is available in the all-pairs computation and will be exploited by our kernel.

1.1.1 All-Pairs N-Body Simulation

We will use gravitiational potential to illustrate the basic form of the computation in an all- pairs N-body simulation. Given N bodies, with an initial position of $\mathbf{x}_i$ and velocities $\mathbf{v}_i$, for $1 \leq i \leq N$. The force vector $\mathbf{f}_{ij}$ on body i caused by its gravitional attraction to body j is given by

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{||\mathbf{r}_{ij}||^2}\frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}$$

where m_i and m_j are the masses of bodies i and j, respectively $\mathbf{r_{ij}} = \mathbf{x_j} - \mathbf{x_i}$ is the vector from body i to body j, and G is the gravitational constant. The left factor, dissolves as the bodies become distant from each other, and the right factor gives a unit vector in the direction of the force (since gravitaional forces are attracting).

The total force on body i can be expressed as

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{f}_{ij} = G m_i \sum_{j \neq i} m_j \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||^3}$$

As bodies get close together, their attractive force grows without bound, since $\mathbf{r}_{i,j} \rightarrow \mathbf{0}$. In astrophysical simulations, collisions between bodies happen. So a *softening factor* $\epsilon^2 > 0$ is added to the denominator. So the total force is re-written as

$$\mathbf{F}_i \approx G m_i \sum_{1 \le j \le N} \frac{m_j \mathbf{r}_{ij}}{\left(||\mathbf{r_{ij}}||^2 + \epsilon^2\right)^{3/2}}$$

With the inclusion of the softening factor the condition of summing over $i \neq j$ is no longer necessary, as $\mathbf{f}_{ii} = 0$ whenever $\epsilon^2 > 0$. The softening factor models the interaction between two Plummer point masses: masses that behave as if they were spherical galaxies (Aarseth 2003, Dyer and Ip 1993). This is desirable for numerical integration of the system state.

To simulate the evolution of the astronomical bodies, we need the acceleration $\mathbf{a}_i = \mathbf{F}_i/m_i$. WE simplify the computation of the acceleration to

$$\mathbf{a}_i \approx G \sum_j \frac{m_j \mathbf{r}_{ij}}{\left(||\mathbf{r}_{ij}||^2 + \epsilon^2\right)^{3/2}}$$

With this we can solve the N-body system using the system of differential equations

$$\ddot{\mathbf{x}}_i = \mathbf{F}_i$$

which is changed to

$$\dot{\mathbf{v}}_i = \mathbf{F}_i$$

$$\dot{\mathbf{x}}_i = \mathbf{v}_i$$

To numerically integrate this system we will use the leapfrog integration scheme (Verlet 1967). The leapfrog scheme uses a "half-step" in time to attain 2nd order accuracy. We can write the algorithm as so:

$$\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \mathbf{a}_i^n \frac{\delta t}{2}$$

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \delta t$$

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \mathbf{a}_i^{n+1} \frac{\delta t}{2}$$

where $\mathbf{a}_i^n = m_i \mathbf{F}(\mathbf{x}_i^n)$. This technique is often used in Gravity simulations since the acceleration is only a function of position. The computational complexity of this operation is $\mathcal{O}(N)$.

1.2 CUDA Implementation of the All-Pairs N-Body Algorithm

We can implement the all-pairs algoirthm as calculating each entry of $\mathbf{f}_{ij}$ in an $N \times N$ grid for all pair-wise forces. Then the total force $\mathbf{F}_i$ on body i, can be calculated by performing a reduction over the j entries. Each entry can be computed indep endently, so there is $\mathcal{O}(N^2)$ parallelism. However, this approach also requires $\mathcal{O}(N^2)$ memeory operations, and would be substantially limited by bandwidth. Instread, some operations are serialized to achieve some data reuse for memory bandwidth.

Like in our matrix multiplication example we will decompose our problem into tiles. Each tile will be a square region containing $p \times p$ elements. Only $2p$ body descriptions are actually required to evaluate all p^2 interactions in the tile (due to symmetry). This will allow us to store these descriptions in shared memory or in thread local registers. The total effect of the interactions in the tile on the p bodies is captured as an udate to p acceleration vectors.

To reuse data, we arrange the computation of a tile so that the interations in each row are evaluated in a squential order, updating the acceleration vector, while the separate rows are done simulataneously. Using this method, we avoid computing all pair-wise forces, storing them just to reduce them. Figure 1 illustrates a schematic figure for our computaitonal tiles.

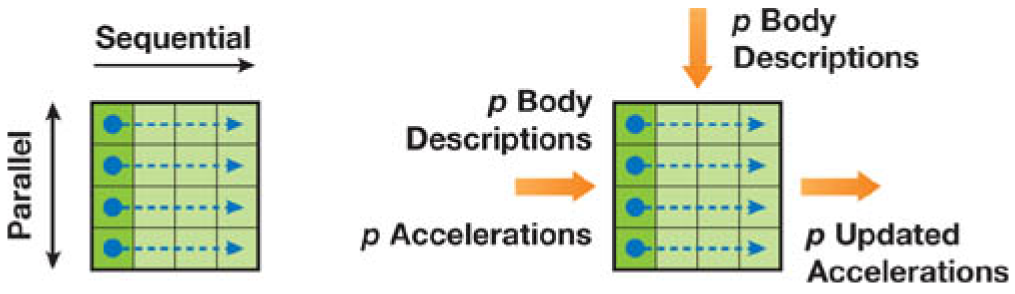


Figure 1: Computational Tile in N-Body solver. GPUGEMS Ch 31

1.2.1 Body to Body Force Calculation

The interaction between two bodies previously described, is implemented in an entirely serial fashion. We will compute the force on body i from its interaction with body j and update the acceleration $\mathbf{a}_i$ of the body i as a result of this interation. There will be 20 floating-point operations in this code snippet, counting the additions, multiplications, a call to `sqrtt()`, and a division (for the reciprocal).

Listing 1: Updating Acceleration of One Body as a result of its interaction with another

```
__device__ void bodyBodyInteraction(float4 bi, float4 bj, float3 &ai)
{

  float3 r;
  // r_ij
  r.x = bj.x - bi.x;
  r.y = bj.y - bi.y;
  r.z = bj.z - bi.z;
  // distSqr = dot(r_ij, r_ij) + EPS^2
   float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;
  // invDistCube =1/distSqr^(3/2)
   float distSixth = distSqr * distSqr * distSqr;
  float invDistCube = 1.0f/sqrtf(distSixth);
  // s = m_j * invDistCube
   float s = bj.w * invDistCube;
  // a_i =  a_i + s * r_ij
  ai.x += r.x * s;
  ai.y += r.y * s;
  ai.z += r.z * s;
}
```

We use CUDA's `float4` data type for the body descritions and accelerations stored on the GPU. We store each body's mass w field of the body's `float4` position. Using a `float4` (instead of `float3`) for the bodies also allows better coalesced memory access to the arrays stored in device memeory, resulting in more efficient memory transactions. Three-dimensional vectors stored in local variables are stored as `float3` variables as register space is an issue while coalescing memory is not.

1.2.2 Calculating Tiles

A tile is processed by p threads performaning a SIMD operation (single instruction multiple data). Each thread will update the acceleration of one body as a result of the interaction with p other bodies. We load p point descriptions, from device memeory into the shared memeory provided to each thread block. Then each thread block evaluates p successive interactions. The result of each tile computation will be p updates accelerations.

The CUDA Code for the tile computation is shown below. The input parameter `myPostiion` holds the information of the body for the executing thread, and the array `shPosition` is an array of body descriptions, held in shared memory. Remember that p threads execute the function body in parallel, and every thread iterates over the same p bodies, thus computing the accelerations of its individual body as a result of p interations.

Listing 2: Interations in a $p \times p$ Tile

```
__device__ void tile_calculation(float4 myPosition, float4 *shPosition,  float3 &accel)
{

  int i;
  for (i = 0; i < blockDim.x; i++) {
    bodyBodyInteraction(myPosition, shPosition[i], accel);
  }
}
```

The architecture supports concurrent reads from multiple threads to a single shared memeory address, so there will be no shared memory bank conflictions during the interaction evaluation. See the *CUDA Programming Guide* for more details.

1.2.3 Clustering Tiles into Thread Blocks

We will have a thread block containing p threads to execute a number of tiles in sequence. Tiles will be sized to balance parallelism with data reuse. This degree of parallelism (that is, the number of rows) will be sufficiently large so that multiple *warps* can be interleaved to hide latencies in the body interaction calculation. The ammount of data reuse grows with the number of columns, and this parameter also governs the size of the transfer of bodies from global memory into shared memory. Note that the size of the tile also determines the register space and shared memory required. We will use square tiles of size $p \times p$. Before computing each tile, each thread moves one body into shared memory. Thus each tile starts with p successive bodies in shared memory.

Figure 2 illustrates a thread block that is executing multple tiles in series. Execution time extends on the horizontal axis, while parallelism spans the vertical. Bolded lines denote the tiles on computation, showing where shared memeory is loaded and a barrier synchronization is performed. In each thread lock, there are N/p tiles, with p threads computing the forces on p bodies. Each thread will compute all N interactions for one body.

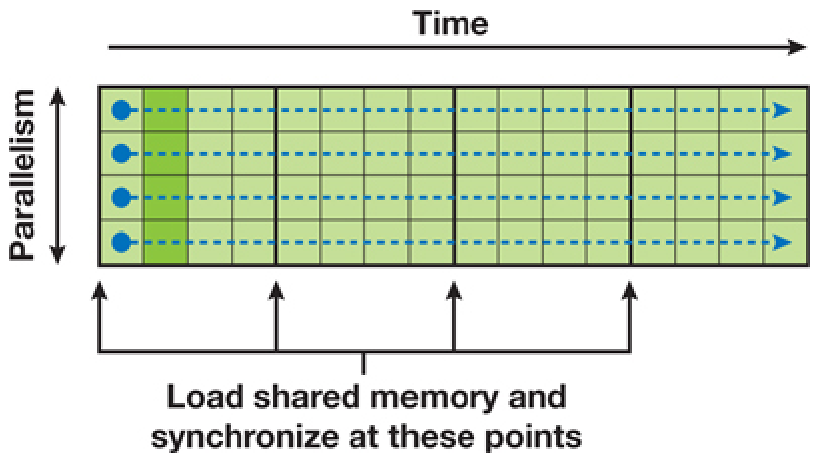


Figure 2: Computation of the tiles as time goes on. GPUGEMS3 CH 31

We write our function `calculate_forces()` to take parameters `d_X` and `d_A`, positions and accelerations of the bodies. The loop over the tiles requires two synchronization points. The first synchronization ensured that all shared memeory locations are populated before the gravitional simulation preceeds. The second barrier ensures that all threads are completed with the tile computation before advancing to the next tile. Without the last synchronization, threads that finish their part in the tile calculation may overwrite the shared memeory still being used by other threads.

Listing 3: Kernel to calculate acceleration via tiles

```
__device__ void calculate_forces(float4 *d_X, float3 *d_A)
{
        __shared__ float4 shPosition[BLOCK_SIZE];
        float4 myPosition;
        int i, tile;
        float3 acc = {0.0f, 0.0f, 0.0f};
        int gtid = blockIdx.x * blockDim.x + threadIdx.x;
        myPosition = d_X[gtid];
        for (i = 0, tile = 0; i < N; i += BLOCK_SIZE, tile++)
        {
                    int idx = tile * blockDim.x + threadIdx.x;
                    shPosition[threadIdx.x] = d_X[idx];
                    __syncthreads();
                    tile_calculation(myPosition, shPosition ,acc);
                    __syncthreads();
          }
    // Save the result in global memory for the integration step.
    d_A[gtid] = acc;
}
```

1.2.4 Defining the Grid of Blocks

The kernel in the above listing calculates the acceleration of p bodies in a system. We will launch this kernel with a one dimensional grid of size N/p in order to calculate the acceleration of all points. We can visualize this process:

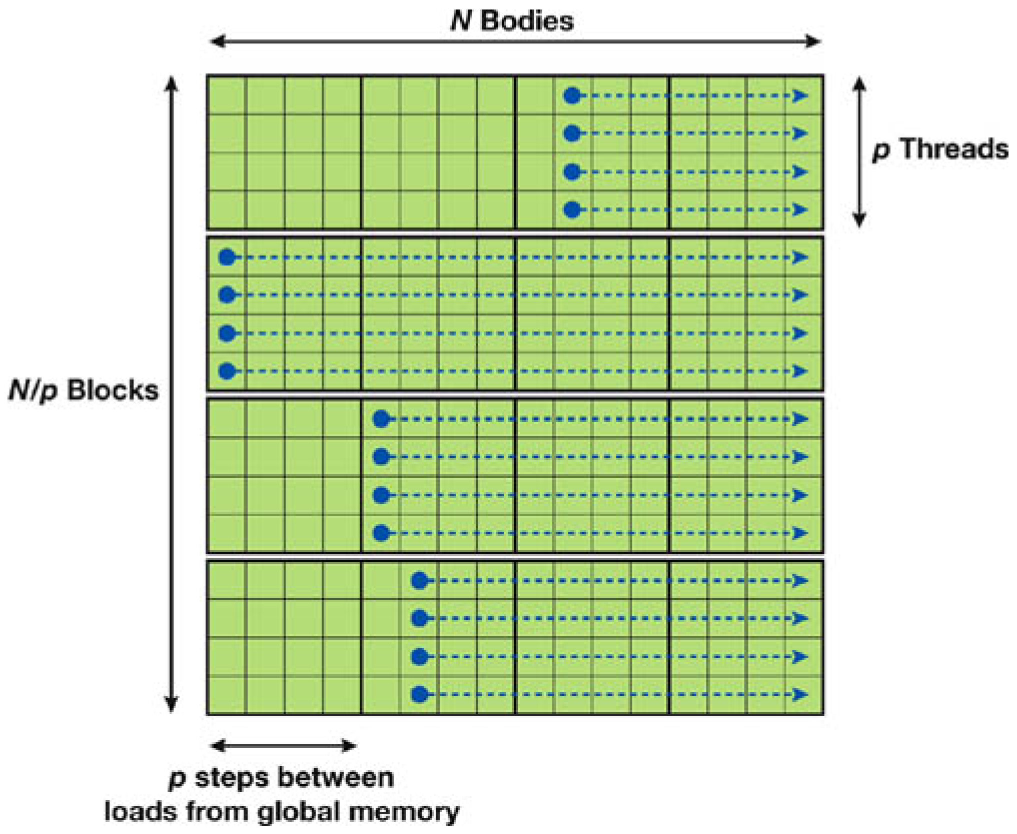


Figure 3: Visual representation of the all-pairs acceleration. GPUGEMS3 CH31

1.3 Implementation of Leapfrog Integration

We have created the means for the force calculation, now we can apply this to our numerical integration scheme. We can reconfigure our numerical scheme to only do two updates

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^n \delta + \frac{1}{2}\mathbf{a}_i^n \delta t^2$$

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{1}{2}\left(\mathbf{a}_i^n + \mathbf{a}_i^{n+1}\right)\delta t$$

This will be our plan for numerical integration

1. Calculate acceleration for $\mathbf{x}_i^n$
2. Use $\mathbf{v}_i^n$ and $\mathbf{a}_i^n$ to calculate $\mathbf{x}_i^{n+1}$
3. Save the old acceleration values and calculate the acceleration for $\mathbf{x}_i^{n+1}$
4. Calculate the velocities for $n+1$

We will output solutions much like we did in the heat equation example. Lets write functions to follow each step.

Listing 4: Calculate $\mathbf{x}_i^{n+1}$

```
__device__ void pos_advance(float4 &X, const float3 V, const float3 A, float dt)
{
        //this is called by each thread
        X.x = X.x + V.x*dt + 0.5f*A.x*dt*dt;
        X.y = X.y + V.y*dt + 0.5f*A.y*dt*dt;
        X.z = X.z + V.z*dt + 0.5f*A.z*dt*dt;
        X.w = X.w; //Mass stays the same
}
```

Listing 5: Calculate $\mathbf{v}_i^{n+1}$

```
__device__ void vel_advance(float3 &V, const float3 A1, const float3 A2, float dt)
{
        //this is called by each thread
        V.x = V.x + 0.5f*(A1.x + A2.z)*dt;
        V.y = V.y + 0.5f*(A1.y + A2.y)*dt;
        V.z = V.z + 0.5f*(A1.z + A2.z)*dt;
}
```

Now that we've written these two functions we can write a kernel to apply one leapfrog step.

Listing 6: Kernel for Leapfrog Stage

```
__global__ void leapfrog(float4 *X, float3 *V, float3 *A, float dt, int k)
{
        if(k==0){ //Initial acceleration.
                calculate_forces(X,A);
                __syncthreads();
        }
        gid = threadIdx.x + blockIdx.x*blockDim.x;
        float3 temp;
        //Store acceleration from x^n
        temp =  A[gid];
        __syncthreads();

        //Calculate x^n+1
        pos_advance(X[gid], V[gid], A[gid], dt);
        __syncthreads();

        //Calculate acceleration at the n+1 stage
        calculate_forces(X,A);
        __syncthreads();

        //Calculate v^n+1
        vel_advance(V[gid], temp, A[gid], dt);
}
```

This kernel computes one update step for the system. We will use a host function to control the stages for N-body simulation.

Listing 7: Function Controlling N-body Simulation

```
void nbody(float4 *X, float dt, int tio, float tend, const int N)
//X are the positions, dt = time step, tio = io iter, tend = end simulation time, N= #of bodies
{
        float4 *d_X;
        float3 *d_A, *d_V;
        float t = 0.0f;
        int k = 0;
        cudaMalloc((void**)&d_X, N*sizeof(float4));
        cudaMalloc((void**)&d_V, N*sizeof(float3));
        cudaMalloc((void**)&d_A, N*sizeof(float3));

        cudaMemcpy(d_X,X, N*sizeof(float4), cudaMemcpyHostToDevice);
        dim3 dimGrid(N/BLOCK_SIZE);
        dim3 dimBlock(BLOCK_SIZE);
        while(t<tmax)
        {
                leapfrog<<<dimGrid,dimBlock>>>(d_X,d_V, d_A, dt, k);
                if(k%tio==0)
                {
                        //IO FUNCTION
                }
                t+=dt;
                k++;
        }
        if(k%tio!=0.0f)
        {
                IO function
        }
        cudaFree(d_X);
        cudaFree(d_A);
        cudaFree(d_V);
}
```

The next figure illustrates the solution of a $N = 2048$ system of bodies, two bodies are 200 times more massive than the rest, we see them pull apart the smaller bodies, until they pull each other twards the center.

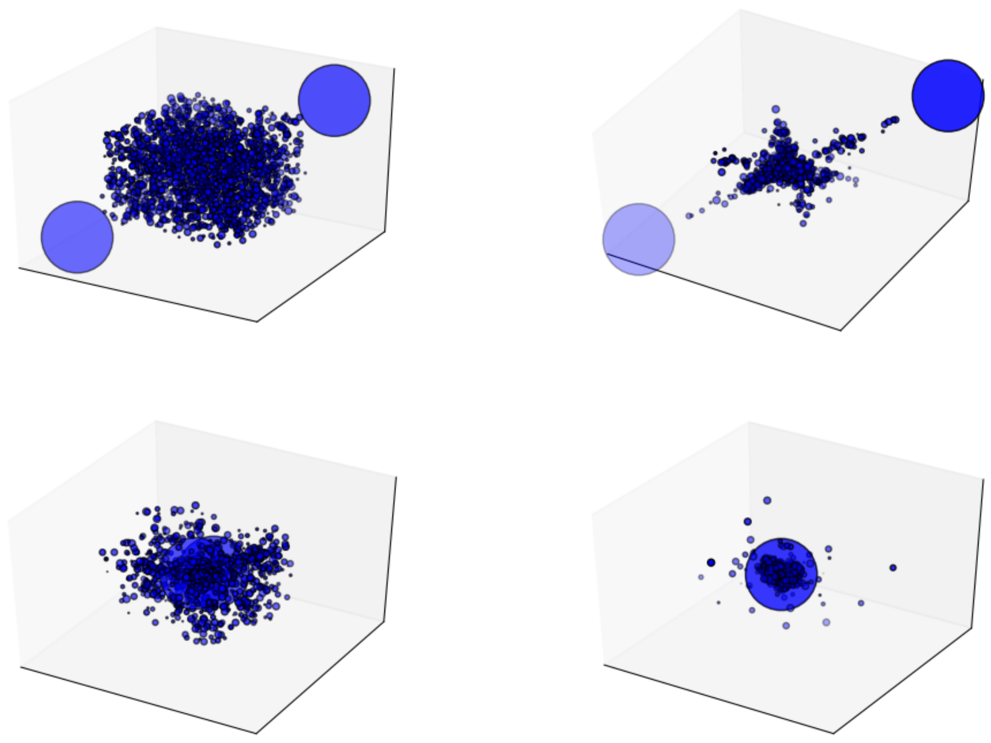


Figure 4: Nbody solution for 2048 bodies, at times $t = 0, 0.3, 0.75, 1$

2 Iterative Methods for Solving Linear Systems

Solving Linear Systems is a corner-stone of many scientific and computing applications. Many direct methods, like Gaussian-Elimination, LU-decomposition are possible on GPU, however, iterative methods are more natural for the GPU's architecture. Additionally, the standard complexity for direct solvers are of the order of N^3, where N is the size of the matrix. Iterative methods *can* be faster, and have a complexity of $\mathcal{O}(MN^2)$ where M is the steps to convergence.

We will consider two algorithms to compute

$$\mathbf{Ax} = \mathbf{b}$$

for a square matrix of size N.

2.1 Gauss-Jacobi Algorithm

We first write

$$\mathbf{A} = \mathbf{D} + \mathbf{R}$$

where $\mathbf{D}$ is a diagonal matrix containing the diagonal elements of $\mathbf{A}$, and $\mathbf{R}$ is the rest. Using this we derive

$$\mathbf{Ax} = (\mathbf{D} + \mathbf{R})\,\mathbf{x} = \mathbf{b} \implies \mathbf{Dx} = \mathbf{b} - \mathbf{Rx} \implies \mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Rx})$$

The inverse of a diagonal matrix is simply a diagonal matrix the reciprocal of the originals elements. Using this derivation we prescribe an iterative method for sourcing the solution $\mathbf{x}$.

$$\mathbf{x}^{n+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^n)$$

This form suggests a matrix multiplication of $\mathbf{D}^{-1}$, which is inefficient, instead, we can write a elementwise formulation of this

$$x_i^{n+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{N} r_{ij} x_j^k\right)$$

If the algorithm converges in a number of steps $M \ll N$ then we can solve $\mathbf{Ax} = \mathbf{b}$ in $\mathcal{O}(MN^2)$ steps instead of $\mathcal{O}(N^3)$ steps, and each newly updated element is data independent of each other. The reason why the if is bolded, is that not all matrices will garauntee a convergence to the real solution. There is a great depth of theory into which matrices will converge, but as a rule of thumb matrices that are *diagonally dominant*. More technically, the Gauss-Jacobi method will converge if $\rho(\mathbf{D}^{-1}\mathbf{R}) < 1$, that is, if the $|\lambda_1| < 1$. These types of matrices are very common in numerical partial differential equations. In addition, the matrix $\mathbf{A}$ is usually banded, so calculating $\mathbf{Rx}$ can be $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$.

Algorithm 1: Psuedocode for Gauss-Jacobi

```
Data: $\mathbf{A} = \mathbf{D} + \mathbf{R}, \mathbf{x}$
1 while $||\mathbf{r}|| > \epsilon$ do
2     $\mathbf{y} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Rx})$;
3     $\mathbf{r} = \mathbf{y} - \mathbf{x}$;
4     $\mathbf{x} = \mathbf{y}$;
```

A serial implementation of the Gauss-Jacobi method will give us insight on it's parallelization:

Listing 8: Gauss-Jacobi for serials

```
void cpu_gauss_jacobi(Matrix A, float *x, float *b, float eps)
{
        float res = 1.0f;
        float summ1, summ2;
        float *temp;
        temp = (float*)malloc(sizeof(x));
        while(res > eps)
        {
                for(int i = 0; i < A.width; i++)
                {
                        summ1 = 0.0f;
                        summ2 = 0.0f;
                        for(int k = 0; k < A.width; k++)
                        {
                                summ1+=A.elements[i+k*A.width]*x[k];
                        }
                        temp[i] = 1/A.elements[i+i*A.width]*(b[i] - summ1];
                        summ2 += temp[i] - x[i];
                }
                res = abs(summ2);
                for(int i = 0; i < A.width; i++)
                        x[i] = temp[i];
        }
}
```

The serial implementation further illustrates that the outer loop shall correspond to the threaded index of our GPU. From this we will create a kernel that performs one iteration of Gauss-Jacobi.

Listing 9: First Try Parallel Gauss Jacobi

```
__global__ void naive_gj(Matrix A, float *x, float* xout, float *b, float eps)
        gid = threadIdx.x + blockIdx.x*blockDim.x;
        float res = 1.0f;
        float summ1 = 0.0f;
        float temp;
        for(int k =0; k< A.width; k++)
        {
                summ1 += A.elements[gid + k*A.width]*x[k];
        }
        temp = 1/A.elements[gid + gid*A.wdith]*(b[gid] - summ1);
        xout[gid] = temp;
}
```

Here's our plan for the parallel Gauss-Jocabi

1. Compute $\mathbf{x}^{n+1}$
2. Calcute $\mathbf{r} = |\mathbf{x}^{n+1} - \mathbf{x}^{n}|$
3. Reduce $\mathbf{r}$ to find the norm of the residual

We can do this by the host function:

Listing 10: Host function to execute Gauss Jacobi plan

```
void par_gj(Matrix A, float *x, float *b, float eps)
{
        float res = 1.0f;
        int counter = 0;
        Matrix d_A;
        d_A.width = A.width;
        d_A.height = A.height;
        float *d_x, *d_b, *d_xnew;
        cudaMalloc((void**)&d_A.elements, A.width*A.height*sizeof(float));
        cudaMalloc((void**)&d_x, A.width*sizeof(float));
        cudaMalloc((void**)&d_b, A.height*sizeof(float));
        cudaMalloc((void**)&d_xnew, A.width*sizeof(float));

        cudaMemcpy(d_A.elements,A.elements,A.width*A.height*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(d_x, x, A.width*sizeof(float),cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, A.height*sizeof(float),cudaMemcpyHostToDevice);

        dim3 dimBlock(16);
        dim3 dimGrid(A.width/dimBlock.x);
        while(res>eps)
        {
                //Compute x^{n+1}
                naive_gj<<<dimGrid,dimBlock>>>(d_A, d_x, d_xnew, d_b, eps);
                cudaDeviceSynchronize();

                //Compute vector of residuals
                compute_r<<<dimGrid,dimBlock>>>(d_x,d_xnew); //Store r in d_x
                cudaDeviceSynchronize();

                //Reduce vector of residuals to find norm
                reduce_r<<<dimGrid,dimBlock>>>(d_x);
                cudaMemcpy(res, d_x, sizeof(float), cudaMemcpyDeviceToHost);
                //X = Xnew
                fill<<<dimGrid,dimBlock>>>(d_x, d_xnew);
                cudaDeviceSynchronize();
                counter++;
                if(counter>A.width)
                        return;
        }
        //export X
        cudaMemcpy(x, d_x, A.width*sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_x);
        cudaFree(d_A.elements);
        cudaFree(d_xnew);
        cudaFree(d_b);
}
```

In order to execute this plan, we need to write `compute_r`, `reduce_r`, `fill`. Fill is a very simple kernel, loading the values of `d_x` from `d_xnew`, and will be left to the reader. We will display `compute_r`, and `reduce_r` can be derived from the chapter on reduction and scan.

Listing 11: Computer the residual vector

```
__global__ void compute_r(d_x,d_xnew)
{
        int gid = threadIdx.x + blockDim.x*blockIdx.x;
        float temp;
        temp = abs(d_xnew[gid]-d_x[gid]);
        d_x[gid] = temp;
}
```

2.2 Gradient Descent

An interesting non-trivial interpretation of the linear system

$$\mathbf{Ax} = \mathbf{b}$$

when $\mathbf{A}$ is a real, positive definite symmetric matrix, is that the solution is also the solution of a minimization problem for a quadratic form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b}$$

To solve this system we calculate the derivative of the function with respect to $\mathbf{x}$,

$$\nabla f = \frac{df}{d\mathbf{x}} = \mathbf{Ax} - \mathbf{b} = 0.$$

This illustrates that the solution to the optimization problem is the solution to the linear system.

The Gradient Descent algorithm is an algorithm that has arised from numerical optimization. This approach updates a test vector $\mathbf{x}$ by applying the gradient of the system. Minimizing f along a particular direction can be done analytically for this quadratic form. Suppose the selected direction vector at step n is $\mathbf{p}^n$, starting from the vector $\mathbf{x}^n$. Then we know that the next iteration is

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \alpha_n\mathbf{p}^n.$$

The key to minimizing the quadratic form is to choose the scalar α_k correctly. Since

$$f(\mathbf{x}^{n+1}) = \frac{1}{2}\mathbf{x}^{n+1}\mathbf{Ax}^{n+1} - \mathbf{x}^{n+1,T}\mathbf{b}$$

$$= \frac{1}{2}\left(\mathbf{x}^n + \alpha_n\mathbf{p}^n\right)^T\mathbf{A}\left(\mathbf{x}^n + \alpha_n\mathbf{p}^n\right) - \left(\mathbf{x}^n + \alpha_n\mathbf{p}^n\right)^T\mathbf{b}$$

$$= \frac{1}{2}\mathbf{x}^{n,T}\mathbf{Ax}^n - \mathbf{x}^{n,T}\mathbf{b}$$

$$+\frac{\alpha_k}{2}\left(\mathbf{p}^{n,T}\mathbf{Ax}^n + \mathbf{x}^{n,T}\mathbf{Ap}^n\right) + \frac{\alpha_k^2}{2}\mathbf{p}^{n,T}\mathbf{Ap}^n - \alpha_k\mathbf{p}^{n,T}\mathbf{b}$$

differentiating this statement by α_k and setting to 0 we find that

$$\alpha_k = \frac{\mathbf{p}^{n,T}\mathbf{r}^n}{\mathbf{p}^{n,T}\mathbf{A}\mathbf{p}^n}$$

where $\mathbf{r}^n = \mathbf{b} - \mathbf{A}\mathbf{x}^n$. This is general for any descent algorithm, for gradient descent, we choose

$$\mathbf{p}^n = -\nabla f(\mathbf{x}^n) = -(\mathbf{A}\mathbf{x}^n - \mathbf{b}) = \mathbf{r}^n.$$

So as per an algorithm, we can write:

Algorithm 2: Gradient Descent for the solution of $\mathbf{Ax} = \mathbf{b}$

```
Data: $\mathbf{A}, \mathbf{x}, \mathbf{b}$
1 $\mathbf{r} = \mathbf{b}$;
2 while $||\mathbf{r}|| > \epsilon$ do
3     $\mathbf{p} = \mathbf{r}$;
4     $\alpha = \frac{\mathbf{p}^T\mathbf{r}}{\mathbf{p}^T\mathbf{A}\mathbf{p}}$;
5     $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$;
6     $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$;
```

Each stage of the gradient descent algorithm requires a matrix-vector multiplication, and is therefore will be order $\mathcal{O}(MN^2)$, much like the Gauss-Jacobi algoirthm, where M is the number of steps to convergence.

In order to realize this algorithm let's form a plan:

1. Compute α
2. Update solution vector
3. Compute gradient for residual and next iteration

In the algorithm, p is always equal to r, so in our actual computation, we will omit it to save memory. Our first order of business is to compute alpha. This will require a couple of steps.

1. Dot product between $\mathbf{p}$ and $\mathbf{r}$
2. Then the dot product between $\mathbf{p}$ and $\mathbf{Ap}$
3. Division of 1 by 2.

To compute a dot product on GPU, we'll employ two operation, a pair-wise multiplication between $\mathbf{p}$ and $\mathbf{r}$, then a reduction of the result. The denominator requires a matrix vector multiplication, and then the dot product described in the numerator.

The algorithm then requires a vector addition, and finally another matrix vector product followed by a vector addition. Therefore we can construct the algorithm based mostly on algorithms we already have.

Listing 12: Host Function for Gradient Descent

```
void grad_descnt(Matrix A, float *x , float *b, float eps)
{
        float *r, *d_b, *d_x, *temp1, *temp2;
        float *alpha, res = 1.0f;
        int counter = 0;
        Matrix d_A;
        dim3 dimBlock(16);
        dim3 dimGrid(A.width/dimBlock.x);

        cudaMalloc(&r, A.width*sizeof(float));
        cudaMalloc(&d_b, A.width*sizeof(float));
        cudaMalloc(&d_x, A.width*sizeof(float));
        cudaMalloc(&d_A.elements, A.width*A.height*sizeof(float));
        cudaMalloc(&alpha, sizeof(float));
        cudaMalloc(&temp1, A.width*sizeof(float));
        cudaMalloc(&temp2, A.width*sizeof(float));

        cudaMemcpy(d_b, b, A.width*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_x, x, A.width*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_A.elements, A.elements, A.width*A.height*sizeof(float), cudaMemcpyHostToDevice);

        while(res > eps)
        {
//      Calculate Alpha
                pairwise_mult<<<dimGrid, dimBlock>>>(r, r, temp1);
                cudaDeviceSynchronize();
                reduce_sum<<<dimGrid, dimBlock>>>(temp1, temp1);
                cudaDeviceSynchronize();
                Matvecmult<<<dimGrid, dimBlock>>>(d_A,r,temp2);
                cudaDeviceSynchronize();
                pairwise_mult<<<dimGrid, dimBlock>>>(r, temp2, temp2);
                cudaDeviceSynchronize();
                reduce_sum<<<dimGrid, dimBlock>>>(temp2,temp2);
                cudaDeviceSynchronize();
                devide<<<1,1>>>(alpha,temp1,temp2);

//      X = X + alpha P
                saxpy<<<dimGrid, dimBlock>>>(d_x, r, alpha);
                cudaDeviceSynchronize();
//      Calculate new r
                Matvecmult<<<dimGrid, dimBlock>>>(d_A, x, temp1);
                cudaDeviceSynchronize();
                saxpy2<<<dimGrid, dimBlock>>>(r, b, temp1, -1);

//      Calculate norm
                vec_abs<<<dimGrid, dimBlock>>>(temp1, r)
                reduce_sum<<<dimGrid, dimBlock>>>(temp1, temp1);
                cudaMemcpy(res, temp1, sizeof(float), cudaMemcpyDeviceToHost);
                counter++;
                if(counter>1e6)
                        return;
        }
        cudaMemcpy(x,d_x,A.width*sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_x);
        cudaFree(d_A.elements);
        cudaFree(r);
        cudaFree(d_b);
        cudaFree(temp1);
        cudaFree(temp2);
        cudaFree(alpha);
}
```

We have done most of the required kernels in previous assignments, and chapters. New ones would be the pairwise multiplication and vector absolute-value, both very simple kernels that the reader is invited to construct.

2.3 Heat Equation Revisited

The heat equation is what is known as a parabolic differential equation. Generally, the type of explicit numerical solution scheme that we illustrated before is not the ideal type of numerical solver for these type of equations. A better type of numerical differential equation scheme is an *implicit* scheme for parabolic equations.

We will consider the **Backward Time, Centered Space Method** (BTCS). Much like the FTCS method invistigated on the chapter on stencil algorithms, BTCS will also rely on a numerical stencil. The difference between FTCS and BTCS, is that the spatial derivative is taken at the $n+1$ time step. So the recurrence relation is

$$\frac{u_j^{n+1} - u_j^n}{\delta t} = \kappa \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_j^{n+1}}{\delta x^2}$$

Here we obtain

$$(1+2r)u_j^{n+1} - ru_{j+1}^{n+1} - ru_{j-1}^{n+1} = u_j^n$$

where $r = \kappa \dfrac{\Delta t}{\Delta x^2}$ This forms a linear system for the solution at $\mathbf{u}^{n+1}$

$$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{u}^n$$

where

$$\mathbf{A} = \begin{pmatrix} 1+2r & -r & 0 & \cdots & 0 \\ -r & 1+2r & -r & \cdots & 0 \\ 0 & -r & 1+2r & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \end{pmatrix}$$

So we need to solve a linear system of a size N_x. This is will be the point for our iterative solvers. The advantage of implicit schemes, is that they are always *numerically stable*, that is theoretically, we can choose any δt that we wish. However, the accuracy of the scheme is $\mathcal{O}(\delta t)$, so we still must be conservative.

We can augment our kernels to take advantage of the tridiagonal matrix.

2.3.1 Modified Gauss-Jacobi

Note that the system is diagonally dominant, so the Gauss-Jocabi algorithm will converge. Here's our new kernel:

Listing 13: Tridiagonal GJ

```
__global__ void tridiag_gj(float *diag, float *subdiag, float *supdiag, float *x, float *xout, float *b,
    float eps)
{
        gid = threadIdx.x + blockDimx.x*blockIdx.x;
        float res = 1.0f;
        float summ1 = 0.0f;
        float temp;

        if(gid>0 && gid < N-1)
        {
                summ1 = subdiag[gid]*x[gid-1] + supdiag[gid]*x[gid+1];
                temp = 1/diag[gid]*(b[gid] - summ1);
        }
        elseif(gid==0)
        {
                summ1 = supdiag[gid]*x[gid+1];
                temp = 1/diag[gid]*(b[gid] - summ1);
        }
        else
        {
                summ1 = subdiag[gid]*x[gid-1];
                temp = 1/diag[gid]*(b[gid] - summ1);
        }
        xout[gid] = temp;
}
```

And then this can be iterated until convergence. Conversely, if we use Gradient descent, we merely need to write a tridiagonal matrix-vector multiplication.

Listing 14: Triadonal Matrix Vector multiplication

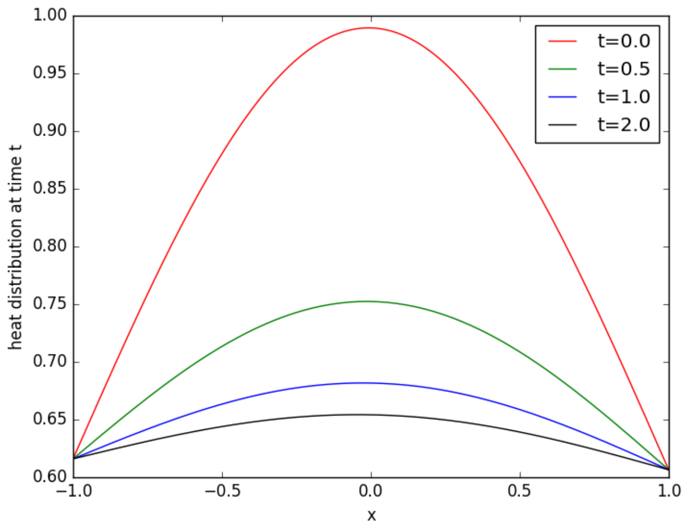
```
template <class T>
__global__ void tridiag_mat_vec(T *diag, T *subdiag, T *supdiag, T *xin, T *xout)
{
        gid = threadIdx.x + blockDim.x*blockIdx.x;
        if(0 < gid && gid < N-1)
                xout[gid] = diag[gid]*xin[gid] + subdiag[gid]*xin[gid-1] + supdiag[gid]*xin[gid+1];
        else if(gid == 0)
                xout[gid] = diag[gid]*xin[gid] + supdiag[gid]*xin[gid+1];
        else
                xout[gid] = diag[gid]*xin[gid] + supdiag[gid]*xin[gid-1];
}
```

You could also create a structure to hold the tridiagonal matrix. This is the only change needed for Gradient Descent. Lets write the routine for simulating the 1D heat equation using BCTS

Listing 15: BTCS

```
void BTCS(float *f, float dt, float dx, float kappa, float tend, int tio)
{
        float r = kappa*dt/(dx*dx);
        float *sub;
        float *diag;
        float *sup;
        float *d_f, *d_f1;
        float t = 0.0f;
        int k =  0;
        float eps = 1e-3;
        std::string f2;
        size_t sz = N*sizeof(float);
        cudaMalloc((void**)&sub, sz);
        cudaMalloc((void**)&diag, sz);
        cudaMalloc((void**)&sup, sz);
        cudaMalloc((void**)&d_f, sz);
        cudaMalloc((void**)&d_f1, sz);
        cudaMemcpy(d_f, f, sz, cudaMemcpyHostToDevice);
        cudaMemcpy(d_f1, f, sz, cudaMemcpyHostToDevice);

        dim3 dimBlock(BLOCK_SIZE);
        dim3 dimGrid(N/BLOCK_SIZE);

        create_tridiag<<<dimGrid,dimBlock>>>(diag, sub, sup, r); //creates tridiag for BTCS
        while(t<tend)
        {
                tridiag_par_gj(diag, sub, sup, d_f1, d_f, eps); //Calculate new value!
                // or
                // tridiag_grad_descent(args);

                fill<<<dimGrid,dimBlock>>>(d_f, d_f1); //u^n+1
                if(k%tio==0)
                {
                        f2 = "sol" + std::to_string(k) + ".dat";
                        cudaMemcpy(f,d_f, sz, cudaMemcpyDeviceToHost);
                        io_fun(f2, f);
                }

                t+=dt;
                k++;
        }
        if(k%tio!=0)
        {
                f2 = "final_sol.dat";
                cudaMemcpy(f,d_f, sz, cudaMemcpyDeviceToHost);
                io_fun(f2, f);
        }

        cudaFree(sub);
        cudaFree(sup);
        cudaFree(diag);
        cudaFree(d_f);
        cudaFree(d_f1);
}
```

Using this prescription we can get the following solution in the fraction of the steps.


t=0.0
t=0.5
t=1.0
t=2.0
heat distribution at time t
x