GPU Programming

Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University
Fall 2025

Optimization Techniques

Dr. Farshad Khunjush

Some slides come from

Dr. Cristina Amza

@ http://www.eecg.toronto.edu/~amza/

And professor Daniel Etiemble

@ http://www.lri.fr/~de/

Returning to Sequential vs. Parallel

- Sequential execution time: t seconds.
- Startup overhead of parallel execution: t_st seconds (depends on architecture)
- (Ideal) parallel execution time: t/p + t_st.
- If t/p + t_st > t, no gain.

General Idea

- Parallelism limited by dependences.
- Restructure code to eliminate or reduce dependences.
- Sometimes possible by compiler, but good to know how to do it by hand.

Optimizations: Example

```
for (i = 0; i < 100000; i++)
        a[i + 1000] = a[i] + 1;
```

Cannot be parallelized as is.

May be parallelized by applying certain code transformations.

Summary

- Reorganize code such that
  - dependences are removed or reduced
  - large pieces of parallel work emerge
  - loop bounds become known
  - …
- Code can become messy … there is a point of diminishing returns.

Factors that Determine Speedup

- Characteristics of parallel code
  - granularity
  - load balance
  - locality
  - communication and synchronization

Granularity

- Granularity = size of the program unit that is executed by a single processor.
- May be a single loop iteration, a set of loop iterations, etc.
- Fine granularity leads to:
  - (positive) ability to use lots of processors
  - (positive) finer-grain load balancing
  - (negative) increased overhead

Granularity and Critical Sections

- Small granularity => more processors => more critical section accesses => more contention.

Issues in Performance of Parallel Parts

- Granularity.
- Load balance.
- Locality.
- Synchronization and communication.

Load Balance

- Load imbalance = different in execution time between processors between barriers.
- Execution time may not be predictable.
  - Regular data parallel: yes.
  - Irregular data parallel or pipeline: perhaps.

Static vs. Dynamic

- Static: done once, by the programmer
  - block, cyclic, etc.
  - fine for regular data parallel
- Dynamic: done at runtime
  - task queue
  - fine for unpredictable execution times
  - usually high overhead
- Semi-static: done once, at run-time

Static Load Balancing

- Block
  - best locality
  - possibly poor load balance
- Cyclic
  - better load balance
  - worse locality
- Block-cyclic
  - load balancing advantages of cyclic (mostly)
  - better locality

Dynamic Load Balancing (1 of 2)

- Centralized: single task queue.
  - Easy to program
  - Excellent load balance
- Distributed: task queue per processor.
  - Less communication/synchronization

Dynamic Load Balancing (2 of 2)

- Task stealing:
  - Processes normally remove and insert tasks from their own queue.
  - When queue is empty, remove task(s) from other queues.
    - Extra overhead and programming difficulty.
    - Better load balancing.

Semi-static Load Balancing

- Measure the cost of program parts.
- Use measurement to partition computation.
- Done once, done every iteration, done every n iterations.

Molecular Dynamics (MD)

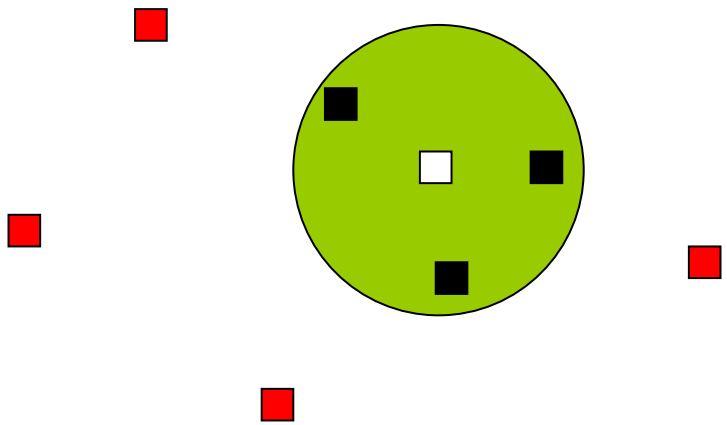
- Simulation of a set of bodies under the influence of physical laws.
- Atoms, molecules, celestial bodies, ...
- Have same basic structure.

Molecular Dynamics (Skeleton)

```
for some number of timesteps {
    for all molecules i
            for all other molecules j
                    force[i] += f( loc[i], loc[j] );
    for all molecules i
            loc[i] = g( loc[i], force[i] );
}
```

Molecular Dynamics

- To reduce amount of computation, account for interaction only with nearby molecules.



Molecular Dynamics (continued)

```
for some number of timesteps {
    for all molecules i
            for all nearby molecules j
                    force[i] += f( loc[i], loc[j] );
    for all molecules i
            loc[i] = g( loc[i], force[i] );
}
```

Molecular Dynamics (continued)

for each molecule i

number of nearby molecules count[i]

array of indices of nearby molecules index[j]

(0 <= j < count[i])

Molecular Dynamics (continued)

```
for some number of timesteps {
    for( i=0; i<num_mol; i++ )
            for( j=0; j<count[i]; j++ )
                    force[i] += f(loc[i],loc[index[j]]);
    for( i=0; i<num_mol; i++ )
            loc[i] = g( loc[i], force[i] );
}
```

Molecular Dynamics (simple)

```
for some number of timesteps {
parallel for
    for( i=0; i<num_mol; i++ )
            for( j=0; j<count[i]; j++ )
                    force[i] += f(loc[i],loc[index[j]]);
parallel for
    for( i=0; i<num_mol; i++ )
            loc[i] = g( loc[i], force[i] );
}
```

Molecular Dynamics (simple)

- Simple to program.
- Possibly poor load balance
  - block distribution of i iterations (molecules)
  - could lead to uneven neighbor distribution
  - cyclic does not help

Better Load Balance

- Assign iterations such that each processor has ~ the same number of neighbors.
- Array of "assign records"
  - size: number of processors
  - two elements:
    - beginning i value (molecule)
    - ending i value (molecule)
- Recompute partition periodically

Frequency of Balancing

- Every time neighbor list is recomputed.
  - once during initialization.
  - every iteration.
  - every n iterations.
- Extra overhead vs. better approximation and better load balance.

Summary

- Parallel code optimization
  - Critical section accesses.
  - Granularity.
  - Load balance.

Some Hints for Vectorization and SIMDization

- Using Pointers avoids Vectorization

```
int a[100];
int *p;
p=a;

for (i=0; i<100;i++)
        *p++ = i;
```

```
int a[100];

for (i=0; i<100;i++)
        p[i] = i;
```

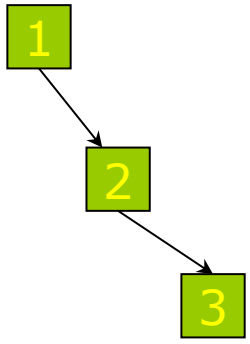
Loop Carried Dependencies

```
S1: A[i]=A[i]+ B[i];
S2: B[i+1]= C[i]+ D[i]
```

```
S2: B[i+1]= C[i]+ D[i]
S1*: A[i+1]=A[i+1]+ B[i+1];
```

Dependencies do not parallelize!

- Dependencies imply sequentiality. They must be broken, if possible, in order to be able to parallelize.
  1. A<-B+C
  2. D<-A*B
  3. E<-C-D



Dependencies do not parallelize!

Privatization

```
do i=1,N
    P: A=...
    Q: X(i)=A+....
end do
```

- In the example above, Q is dependent on P, and because of this, the loop cannot be parallelized.
- Assuming that there is no circular dependence of P on to Q, the privatization method helps break this dependence.

```
pardo i=1,N
    P: A(i)=...
    Q: X(i)=A(i)+...
end pardo
```

Dependencies do not parallelize!

- In OpenMP, similar results could be achieved if A were to be declared private.

```
#pragma omp parallel for private(A)
for( i=0; i<N; i++) {
   A =...                    ;
   X[i]=A +...          ;
   }
```

Dependencies do not parallelize!

- In OpenMP, if explicit privatization is used, then

```
#pragma omp parallel for
for( i=0; i<N; i++) {
   A[i]=...          ;
   X[i]=A[i]+...     ;
   }
```

Dependencies do not parallelize!

Reduction

```
do i=1,N
    P: X(i)=...
    Q: Sum=Sum+X(i)
end do
```

Statement Q depends on itself since the sum is built sequentially. This type of calculation can be parallelized depending on the underlying system. For example, if the underlying system is a shared memory one, one can easily derive the sum in $\log_2 N$ time (provided that there are enough processors to carry out N/2 additions in parallel).

Dependencies do not parallelize!

```
pardo i=1,N
    P: X(i)=...
    Q: Sum=sum_reduce(X(i))
end pardo
```

Dependencies do not parallelize!

Induction

If a loop depicts a recursion on one of the variables where to calculate the result of one iteration, one needs the result of a previous e.g.
$x(i)=x(i\text{-}1)+y(i)$

one can use the carry generation and propagation techniques (i.e. solving the recursion) in order to parallelize the code.

This method is called induction.

Memory Access Pattern

- All the elements of the line are used before the next line is referenced.
- This type of access pattern is often referred to as "unit stride."

```
for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
                sum += a[i][j];
```

V

```
for (int j=0; j<n; j++)
        for (int i=0; i<n; i++)
                sum += a[i][j];
```

NV

Memory Access Pattern

```
void loop_interchange_example(float *a, float *b, float *c)
{
        for(int j=0; j<100; j++) {                    NV
                for(int i=0; i<100; i++) {
                        a[i,j] = a[i,j]+ b[i,j]*c[i,j];}}}
```

```
void loop_interchange_example(float *a, float *b, float *c)
{
        for(int i=0; i<100; i++) {                    V
                for(int j=0; j<100; j++) {
                        a[i,j] = a[i,j]+ b[i,j]*c[i,j];}}}
```

Loop Spliting

```
void splitting_example(float *a, float *b, float *c, float *d, float *e, float *f, float *g)
{
    for(int i=0; i<99; i++) {
            a[i] = c[i] + e[i] * b[i];
            b[i] = x + a[i] + g[i];
            c[i+1] = a[i] + b[i] + f[i];
    }
}
```

```
for (int i=0; i<99; i++) {
            d[i] = e[i] * b[i];
}
for(int i=0; i<99; i++) {
            a[i] = c[i] + d[i];
            b[i] = x + a[i] + g[i];
            c[i+1] = a[i] + b[i] + f[i];
}
```

Loop Optimizations

- If any memory location is referenced more than once in the loop nest and if at least one of those references modifies its value, then their relative ordering must not be changed by the transformation.

Loop Optimizations

- *Loop unrolling:* to effectively reduce the overheads of loop execution

```
for (int i=1; i<n; i++) {
        a[i] = b[i] + 1;
        c[i] = a[i] + a[i-1] + b[i-1];
}
```

```
for (int i=1; i<n; i+=2) {
        a[i] = b[i] + 1;
        c[i] = a[i] + a[i-1] + b[i-1];
        a[i+1] = b[i+1] + 1;
        c[i+1] = a[i+1] + a[i] + b[i];}
```

Loop Optimizations

- *Loop unrolling:* to effectively reduce the overheads of loop execution

```
for (int j=0; j<n; j++)
        for (int i=0; i<n; i++)
                a[i][j] = b[i][j] + 1;
```

```
for (int j=0; j<n; j++)
        for (int i=0; i<n; i+=2){
                a[i][j] = b[i][j] + 1;
                a[i+1][j] = b[i+1][j] + 1;}
```

Strided Access

Loop Optimizations

```
for (int j=0; j<n; j+=2){
        for (int i=0; i<n; i++)
                a[i][j] = b[i][j] + 1;
        for (int i=0; i<n; i++)
                a[i][j+1] = b[i][j+1] + 1;
}
```

Unroll & jam

```
for (int j=0; j<n; j+=2){
        for (int i=0; i<n; i++){
                a[i][j] = b[i][j] + 1;
                a[i][j+1] = b[i][j+1] + 1;
        }
}
```

Loop Optimization

```
void loop_fusion_example(float *a, float *b, float *c, float *d)
{
  for(int i=0; i<100; i++)
        a[i] = b[i] + c[i];
  for(i=0; i<99; i++)
        d[i] = a[i] * 2;
}


  for(i=0; i<99; i++) {
        a[i] = b[i] + c[i];
        d[i] = a[i] * 2;
  }
  a[99] = b[99] + c[99];
```

Loop Optimization (Loop Fission)

```
for (int i=0; i<n; i++)
{
        e[i] = exp(i/n) ;
        for (int j=0; j<m; j++)
                a[j][i] = b[j][i] + d[j] * e[i];
}
```

```
for (int i=0; i<n; i++)
        e[i] = exp(i/n) ;

for (int j=0; j<m; j++)
        for (int i=0; i<n; i++)
                a[j][i] = b[j][i] + d[j] * e[i];
```

Loop Fission

```
for(int i=0; i<100; i++) {
        a[i] = (b[i] + b[i+1])/2;
        b[i+1] = c[i];
}
```

```
for(i=0; i<100; i++)
        d[i] = b[i+1];
for(i=0; i<100; i++) {
        a[i] = (b[i] + d[i])/2;
        b[i+1] = c[i];
}
```

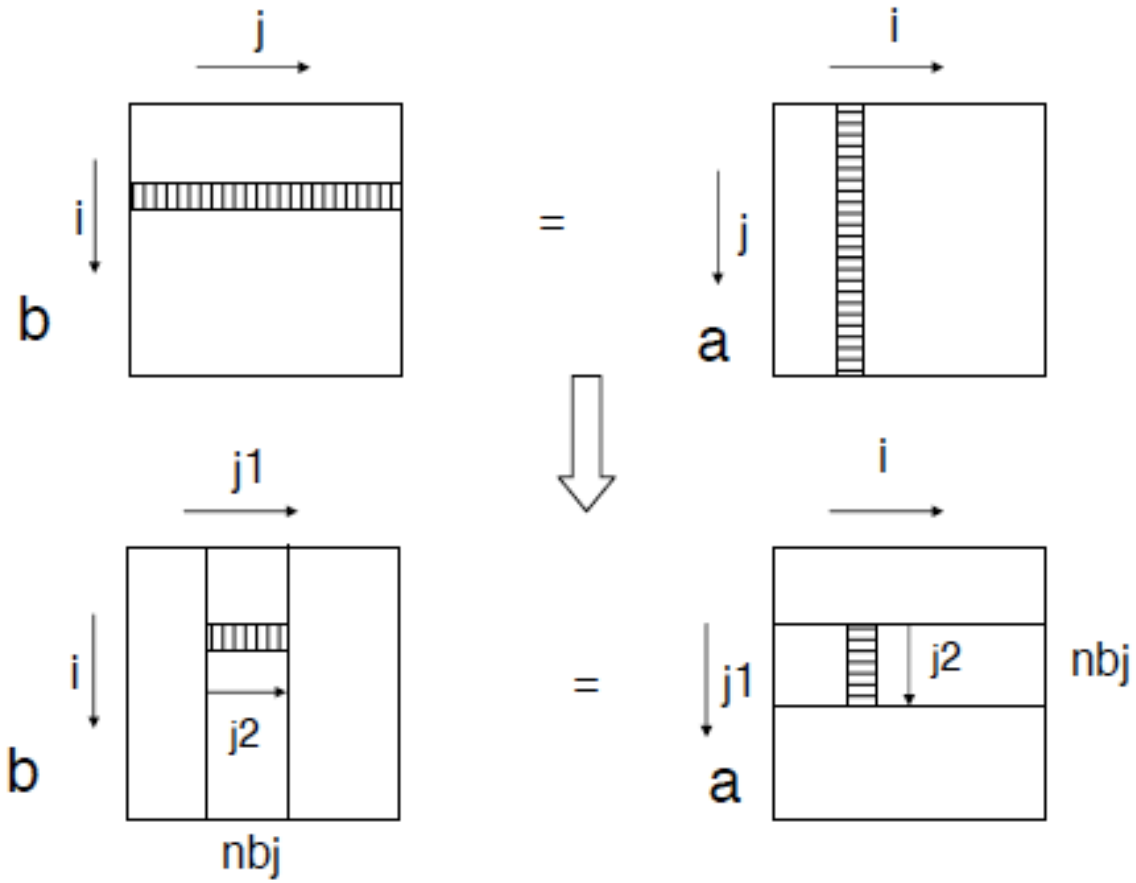
```
for(i=0; i<100; i++)
        d[i] = b[i+1];
a[0] = (b[0] + d[0])/2
for(i=0; i<99; i++) {
        b[i+1] = c[i];
        a[i+1] = (b[i+1] + d[i+1])/2;
}
B[100] = c[99];
```

Loop Optimization (Loop tiling or blocking)

```
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        b[i][j] = a[j][i];
```

```
for (int j1=0; j1<n; j1+=nbj)
   for (int i=0; i<n; i++)
     for (int j2=0; j2 < MIN(n-j1,nbj); j2++)
         b[i][j1+j2] = a[j1+j2][i];
```

Loop Optimization (Loop tiling or blocking)



Considerations in Using OpenMP

- Optimize Barrier Use
  - they are expensive operations → Reduce using it

```
#pragma omp parallel
{
    .........
    #pragma omp for
    for (i=0; i<n; i++)
       .........
    #pragma omp for nowait
    for (i=0; i<n; i++)
 } /*-- End of parallel region - barrier is implied --*/
```

Optimize Barrier Use

```
#pragma omp parallel default(none) \
        shared(n,a,b,c,d,sum) private(i)
{
        #pragma omp for nowait
        for (i=0; i<n; i++)
                a[i] += b[i];

        #pragma omp for nowait
        for (i=0; i<n; i++)
                c[i] += d[i];

        #pragma omp barrier
        #pragma omp for nowait reduction(+:sum)
        for (i=0; i<n; i++)
                sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Avoid Large Critical Regions

```
#pragma omp parallel shared(a,b) private(c,d)
{
        ......
        #pragma omp critical
        {
                a += 2 * c;
                c = d * d;
        }
} /*-- End of parallel region --*/
```

Considerations in Using OpenMP (Cont'd)

- Maximize Parallel Regions
  - Overheads are associated with starting and terminating a parallel region
  - Large parallel regions offer more opportunities for using data in cache and provide a bigger context for other compiler optimizations
    - Minimize the number of parallel regions

Maximize Parallel Regions

```
#pragma omp parallel for
for (.....)
{
        /*-- Work-sharing loop 1 --*/
}
#pragma omp parallel for
for (.....)
{
        /*-- Work-sharing loop 2 --*/
}
        .........
#pragma omp parallel for
for (.....)
{
        /*-- Work-sharing loop N --*/
}
```

```
#pragma omp parallel
{
        #pragma omp for
/*-- Work-sharing loop 1 --*/
        { ..... }

        #pragma omp for
/*-- Work-sharing loop 2 --*/
        { ..... }
.........
        #pragma omp for
/*-- Work-sharing loop N --*/
        { ..... }
}
```

Avoid Parallel Regions in Inner Loops

```
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
                #pragma omp parallel for
                for (k=0; k<n; k++)
                { .........}
```

```
#pragma omp parallel
        for (i=0; i<n; i++)
                for (j=0; j<n; j++)
                        #pragma omp for
                        for (k=0; k<n; k++)
                        { .........}
```

Address Poor Load Balance

- Threads might have different amounts of work to do
  - The threads wait at the next synchronization point until the slowest one completes → Use Schedule Clause

```
for (i=0; i<N; i++) {
        ReadFromFile(i,...);
        for (j=0; j<ProcessingNum; j++ )
                ProcessData(); /* lots of work here
*/
        WriteResultsToFile(i);
}
```

Address Poor Load Balance

```
#pragma omp parallel
{
        /* preload data to be used in first iteration of the i-loop */
        #pragma omp single
        {ReadFromFile(0,...);}
        for (i=0; i<N; i++) {
                /* preload data for next iteration of the i-loop */
                #pragma omp single nowait
                {ReadFromFile(i+1...);}
                #pragma omp for schedule(dynamic)
                for (j=0; j<ProcessingNum; j++)
                        ProcessChunkOfData(); /* here is the work */
                /* there is a barrier at the end of this loop */
                #pragma omp single nowait
                {WriteResultsToFile(i);}
        } /* threads immediately move on to next iteration of i-loop */
} /* one parallel region encloses all the work */
```

Avoid False Sharing

- A side effect of the cache-line granularity of cache coherence implemented in shared-memory systems

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
for (int i=0; i<Nthreads; i++)
        a[i] += i;
```

Nthreads = 8 → a[1] through a[7] are in the same cache line

array padding can be used to eliminate the problem

by dimensioning the array as a[n][8]
changing the indexing from a[i] to a[i][0] eliminates the

Optimizing FFT function

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{COMPLEX u,temp,tm;
    int i,j,le,windex;
    windex = 1;
    for(le=n/2 ; le > 0 ; le/=2) {
        wptr = w;
        for (j = 0 ; j < le ; j++) {
            u = *wptr;
            for (i = j ; i < n ; i = i + 2*le) {
                xi = x + i;
                xip = xi + le;
                temp.real = xi->real + xip->real;
                temp.imag = xi->imag + xip->imag;
                tm.real = xi->real - xip->real;
                tm.imag = xi->imag - xip->imag;
                xip->real = tm.real*u.real - tm.imag*u.imag;
                xip->imag = tm.real*u.imag + tm.imag*u.real;
                *xi = temp; }
            wptr = wptr + windex;}
        windex = 2*windex;}}
```

Slides come from Professor Daniel Etiemble

Optimizing FFT function

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{
     COMPLEX u,temp,tm;
     int i,j,le,windex;
     windex = 1;
     for(le=n/2 ; le > 0 ; le/=2) {
            for (j = 0 ; j < le ; j++) {
              k=0;
              for (i = j ; i < n ; i = i + 2*le) {
                   tm.real = x[i].real - x[i+le].real;
                   tm.imag = x[i].imag - x[i+le].imag;
                  x[i+le].real = tm.real*w[k].real - tm.imag* w[k].imag;
                  x[i+le].imag;= tm.real*w[k].imag + tm.imag* w[k].real;
                  x[i].real = x[i].real + x[i+le].real;
                  x[i].imag = x[i].imag + x[i+le].imag; }
              k += windex;
            }
            windex = 2*windex }
```

Optimizing FFT function

```
void fft_c(int n, float x_r[],float x_i, float w_r[], float w_i[])
{
   register float tm_r, tm_i;
   int i,j,le,windex;
   windex = 1;
   for(le=n/2 ; le > 0 ; le/=2) {
      for (i = 0 ; i < n ; i = i + 2*le) {
         for (j = 0 ; j < le ; j++){
            tm_r = x_r[i+j] – x_r[i+j+le];
            tm_i = x_i[i+j]– x_i[i+j+le];
            x_r[i+j] = x_r[i+j] + x_r[i+j+le];
            x_i[i+j] = x_i[i+j] + x_i[i+j+le];
            x_r[i+j+le] = tm_r*w_r[j*windex] – tm_i* w_i[j*windex];
            x_i[i+j+le] = tm_r*w_i[j*windex] + tm_i* w_r[j*windex];
         }
      }
      windex = 2*windex;
   }}
```

Optimizing FFT function

```
void fft_c(int n, float x_r[], float x_i[], float w_r[], float w_i[])
{
   register float t_r,t_i;
   int i,j,le,windex, k;
   windex = 1;k=0;
   for(le=n/2 ; le > 0 ; le/=2,k++) {
       for (i = 0 ; i < n ; i = i + 2*le) {
           for (j = 0 ; j < le ; j++) {
               t_r=x_r[i+j]-x_r[i+j+le];
               t_i=x_i[i+j]-x_i[i+j+le];
               x_r[i+j]=x_r[i+j]+x_r[i+j+le];
               x_i[i+j]=x_i[i+j]+x_i[i+j+le];
               x_r[i+j+le]=t_r*w_r[(n/2)*k+j]-t_i*w_i[(n/2)*k+j];
               x_i[i+j+le]=t_r*w_i[(n/2)*k+j]+t_i*w_r[(n/2)*k+j];}}
     windex = 2*windex
   }}
```