Name: _____

Directions: **Work only on this sheet** (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing.

Unless otherwise stated, give numerical answers as expressions, e.g. $\frac{2}{3} \times 6 - 1.8$. Do NOT use calculators.

1. (35) The code below implements the Jacobi algoritm in OpenMP. Fill in the blanks, and add any lines necessary. For the latter action, write something like, "Place the following code between lines 8 and 9." Do NOT delete or change lines.

```
1  #include <omp.h>
2
3  // partitions s..e into nc chunks, placing the
4  // ith in first and last
5  void chunker(int s, int e, int nc, int i,
6     int *first, int *last)
7  {  int chunksize = (e-s+1) / nc;
8     *first = s + i * chunksize;
9     if (i < nc-1) *last = *first + chunksize - 1;
10    else *last = e;
11 }
12
13 // returns the "dot product" of vectors u and v
14 float innerprod(float *u, float *v, int n)
15 {  float sum = 0.0; int i;
16    for (i = 0; i < n; i++)
17       sum += u[i] * v[i];
18    return sum;
19 }
20
21 // solves AX = Y, A nxn; stop iteration when
22 // total change is < n*eps
23 void jacobi(float *a, float *x, float *y, int n, float eps)
24 {
25    float *oldx = malloc(n*sizeof(float));
26    float se;
27    #pragma omp parallel
28    {  int i;
29       int thn = omp_get_thread_num();
30       int nth = omp_get_num_threads();
31       int first,last;
32       chunker(0,n-1,nth,thn,&first,&last);
33       for (i = first; i <= last; i++) oldx[i] = x[i] = 1.0;
34       float tmp;
35       while (1) {
36          for (i = first; i <= last; i++) {
37             tmp = innerprod(_____)
38             tmp -= a[n*i+i] * oldx[i];
39             _____;
40          }
41          for (i = first; i <= last; i++)
42             se += abs(x[i]-oldx[i]);
43          _____;
44          for (i = first; i <= last; i++)
45             oldx[i] = x[i];
46       }
47    }
48 }
```

2. (35) The code below implements the odd/even transposition sort in CUDA. Fill in the blanks, and add any lines necessary. For the latter action, write something like, "Place the following code between lines 8 and 9." Do NOT delete or change lines.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  // compare and swap, thread number "me"; copies from f to t,
6  // swapping f[i] and f[j] if the higher-index value is smaller;
7  // it is required that i < j
8  _____cas(int *f,int *t,int i,int j, int n, int me)
9  {
10    if (i < 0 || j >= n) return;
11    if (me == i) {
12       if (f[i] > f[j]) t[me] = f[j];
13       else t[me] = f[i];
14    } else {  // me == j
15       if (f[i] > f[j]) t[me] = f[i];
16       else t[me] = f[j];
17    }
18 }
19
20 // does one iteration of the sort
21 __global__ void oekern(int *da, int *daaux, int n, int iter)
22 {  int bix = _____;
23    if (iter % 2) {
24       if (bix % 2) cas(da,daaux,bix-1,bix,n,bix);
25       else cas(da,daaux,bix,bix+1,n,bix);
26    } else {
27       if (bix % 2) cas(da,daaux,bix,bix+1,n,bix);
28       else cas(da,daaux,bix-1,bix,n,bix);
29    }
30 }
31
32 // sorts the array ha, length n, using odd/even transp. sort;
33 // kept simple for illustration, no optimization
34 void oddeven(int *ha, int n)
35 {
36    int *da;
37    int dasize = n * sizeof(int);
38    cudaMalloc((void **)&da,dasize);
39    cudaMemcpy(da,ha,dasize,cudaMemcpyHostToDevice);
40    // the array daaux will serve as "scratch space," a copy of da; will
41    // be using cudaMemcpy() with cudaMemcpyDeviceToDevice argument to
42    // copy back
43    // we need daaux because the hardware lacks _____
44    // _____
45    int *daaux;
46    cudaMalloc((void **)&daaux,dasize);
47    cudaMemcpy(daaux,ha,dasize,cudaMemcpyHostToDevice);
48    dim3 dimGrid(n,1);
49    dim3 dimBlock(1,1,1);
50    for (int iter = 1; iter <= n; iter++) {
51    }
52    cudaMemcpy(ha,da,dasize,cudaMemcpyDeviceToHost);
53 }
```

3. A **wavefront** operation on an nxn matrix A works on diagonals. Here we will define the ith "wave" to consist of $a_{i0}, a_{i-1,1}, ..., a_{0,i}$. (Note that these diagonals are perpendicular to the usual ones.) Suppose we are developing a highly parallel implementation, say in CUDA. Say we set up the calculation for wave i so that a different thread handles each element in the wave. So, assigning from "southwest to northeast," thread 0 would handle a_{i0}, thread 1 would handle $a_{i-1,1}$, and so on.

(a) (15) Fill in the blanks: An obvious problem is that some threads have less work to do than others. In standard terminology, we say that this is a _____ *problem.* Thread i will only be busy a fraction _____ of the time, i = 0,1,...,n-1.

(b) (15) Suppose n = 7 and shared memory has 8 banks. For which values of i will the ith wave generate no bank conflicts?

Solutions:

1.

```
#include <omp.h>

// partitions s..e into nc chunks, placing the ith in first and last (i
// = 0,...,nc-1)
void chunker(int s, int e, int nc, int i, int *first, int *last)
{  int chunksize = (e-s+1) / nc;
   *first = s + i * chunksize;
   if (i < nc-1) *last = *first + chunksize - 1;
   else *last = e;
}

// returns the "dot product" of vectors u and v
float innerprod(float *u, float *v, int n)
{  float sum = 0.0; int i;
   for (i = 0; i < n; i++)
      sum += u[i] * v[i];
   return sum;
}

// solves AX = Y, A nxn; stop iteration when total change is < n*eps
void jacobi(float *a, float *x, float *y, int n, float eps)
{
   float *oldx = malloc(n*sizeof(float));
   float se;
   #pragma omp parallel
   {  int i;
      int thn = omp_get_thread_num();
      int nth = omp_get_num_threads();
      int first,last;
      chunker(0,n-1,nth,thn,&first,&last);
      for (i = first; i <= last; i++) oldx[i] = x[i] = 1.0;
      float tmp;
      while (1) {
         for (i = first; i <= last; i++) {
            tmp = innerprod(&a[n*i],oldx,n);
            tmp -= a[n*i+i] * oldx[i];
            x[i] = (y[i] - tmp) / a[n*i+i];
         }
         #pragma omp barrier
         #pragma omp for reduction(+:se)
         for (i = first; i <= last; i++)
            se += abs(x[i]-oldx[i]);
         #pragma omp barrier
         if (se < n*eps) break;
         for (i = first; i <= last; i++)
            oldx[i] = x[i];
      }
   }
}
```

2. Some students pointed out that swapping **da** and **daaux** was better than copying. This is actually what I intended in the first place, but during debugging I "temporarily" change it to a copy, which was easier. I then forgot to change it back later.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// compare and swap; copies from the f to t, swapping f[i] and
// f[j] if the higher-index value is smaller; it is required that i < j
__device__ void cas(int *f,int *t,int i,int j, int n, int me)
{
   if (i < 0 || j >= n) return;
   if (me == i) {
      if (f[i] > f[j]) t[me] = f[j];
      else t[me] = f[i];
   } else {  // me == j
      if (f[i] > f[j]) t[me] = f[i];
      else t[me] = f[j];
   }
}

// does one iteration of the sort
__global__ void oekern(int *da, int *daaux, int n, int iter)
{  int bix = blockIdx.x;  // block number within grid
   if (iter % 2) {
      if (bix % 2) cas(da,daaux,bix-1,bix,n,bix);
      else cas(da,daaux,bix,bix+1,n,bix);
   } else {
      if (bix % 2) cas(da,daaux,bix,bix+1,n,bix);
      else cas(da,daaux,bix-1,bix,n,bix);
   }
}

// sorts the array ha, length n, using odd/even transp. sort;
// kept simple for illustration, no optimization
void oddeven(int *ha, int n)
{
   int *da;
   int dasize = n * sizeof(int);
   cudaMalloc((void **)&da,dasize);
   cudaMemcpy(da,ha,dasize,cudaMemcpyHostToDevice);
   // the array daaux will serve as "scratch space," a copy of da; will
   // be using cudaMemcpy() with cudaMemcpyDeviceToDevice argument to
   // copy back
   // we need daaux because the hardware lacks facilities for
   // synchronization between blocks
   int *daaux;
   cudaMalloc((void **)&daaux,dasize);
   cudaMemcpy(daaux,ha,dasize,cudaMemcpyHostToDevice);
   dim3 dimGrid(n,1);
   dim3 dimBlock(1,1,1);
   for (int iter = 1; iter <= n; iter++) {
      oekern<<<dimGrid,dimBlock>>>(da,daaux,n,iter);
      cudaThreadSynchronize();
      cudaMemcpy(da,daaux,dasize,cudaMemcpyDeviceToDevice);
   }
   cudaMemcpy(ha,da,dasize,cudaMemcpyDeviceToHost);
}
```

3. This problem was slightly misspecified, which made it a little easier. There really should be 2n-1 waves, not just n.

(a) This is a **load balancing** problem. Thread i will be busy only a fraction (n-i)/n of the time.

(b) Without loss of generality, assume that the array starts at word address 0. Note that consecutive elements within a wave are 6 words apart. Then we have the following table:

wave	addresses	banks
0	0	0
1	1,7	1,7
2	2,8,14	2,0,6
3	3,9,15,21	3,1,7,5
4	4,10,16,22,28	4,2,0,6,4
5	5,11,17,23,29,35	5,3,1,7,5,3
6	6,12,18,24,30,36,42	6,4,2,0,6,4,2

Only threds 0, 1, 2 and 3 avoid bank conflicts.