Name: ....

Université de Lille1 *Master 2 - Calcul Scientifique, 2010-2011*

Exam of Supercomputing
Duration: 3H - Authorized Ducuments

Partie II - N. Melab & T-V. Luong

Exercise 1 - Grid Computing

The design of parallel distributed applications on computational grids requires to take into account the characteristics of these environments such as the volatility and heterogeneous nature of the computational resources. In this exercise, the focus is on the design of a Grid-based parallel distributed algorithm to compute a matrix-vector product using Grid'5000. The matrix and vector are assumed to be very large so that a Grid is required to compute their product.

1. The heterogeneity of the Grid means here that the computational resources have different computing powers in terms of FLOPS. Without giving the technical details, explain how to design a Grid-enabled parallel distributed algorithm to compute a matrix-vector product taking into account the heterogeneous nature of resources.

2. In the following questions, the issue of volatility of resources is addressed. Volatility means that the computational resources may join or leave the grid dynamically at any moment.

   (a) What are the causes of the volatility of resources in Grid'5000?

   (b) Without giving the technical details, explain how to design a Grid-based parallel distributed algorithm to compute a matrix-vector product taking into account the volatility of resources.

3. The speed-up metric is traditionally used to evaluate the performance in term of efficiency of parallel applications. In a static (non volatile) and homogeneous environment, the speed-up of a parallel application is defined as the ratio between the sequential execution time (on a single processor) of the application on the parallel execution time (on several processors) of the application.

   The traditional way to compute the speed-up is not well-suited to heterogeneous and volatile computational grids. Why?

Exercise 2 - GPU Computing using CUDA

1. The CUDA host-side code given below compiles successfully but provides wrong results at runtime. Find and correct the error(s).

```
void RandomInit(float* data, int n) {
  for (int i = 0; i < n; ++i)
    data[i] = rand() / (float)RAND_MAX;
}

int main(int argc, char** argv) {
  int N = 50000;
  size_t size = N;
```

```
  float* h_A, float* h_B, float* h_C;
  float* d_A, float* d_B, float* d_C;

  h_A = (float*)malloc(size);
  h_B = (float*)malloc(size);
  h_C = (float*)malloc(size);

  RandomInit(h_A, N);
  RandomInit(h_B, N);

  cudaMalloc((void**)&d_A, size);
  cudaMalloc((void**)&d_B, size);
  cudaMalloc((void**)&d_C, size);

  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost) ;
  ...

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  free(h_A); free(h_B); free(h_C);
}
```

.....
.....
.....
.....

2. In the CUDA code given below, suppose that macros are given. The program gives incorrect results at runtime. Find and correct the error(s).

```
void randomInit(int* data, int size) {
  for (int i = 0; i < size; ++i)
    data[i] = rand()%20;
}

int main (int argc, char** argv) {

  unsigned int size_A = WA * HA;
  unsigned int mem_size_A = sizeof(int) * size_A;
  int* h_A = (int*) malloc(mem_size_A);
  unsigned int size_B = WB * HB;
  unsigned int mem_size_B = sizeof(int) * size_B;
  int* h_B = (int*) malloc(mem_size_B);

  randomInit(h_A, size_A);
  randomInit(h_B, size_B);
```

```
  int* d_A;
  cudaMalloc((void**) &d_A, mem_size_A);
  int* d_B;
  cudaMalloc((void**) &d_B, mem_size_B);

  cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, mem_size_B,cudaMemcpyHostToDevice);

  unsigned int size_C = WC * HC;
  unsigned int mem_size_C = sizeof(int) * size_C;
  int* d_C = (int*) malloc(mem_size_C);

  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid(WC / threads.x, HC / threads.y);

  matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);

  int* reference = (int*) malloc(mem_size_C);

  cudaMemcpy(reference, d_C, mem_size_C, cudaMemcpyDeviceToHost);
  ...
  cudaFree(d_A); cudaFree(d_B);
  free(h_A); free(h_B); free(reference);
}
```

.....
.....
.....
.....
.....

3. The efficiency of a CUDA program depends strongly on two major parameters: the total number of threads and the number of threads per block. The objective here is to propose an approach that allows the automatic tuning of these two parameters for an iterative program. The approach is very simple and consists in using different values of these parameters during the first iterations of the program. Once the best values for the two parameters are found they are used for all the following iterations. The CUDA host-side code given below is supposed to perform this task. Briefly explain why this host-side code may not provide a good tuning. Correct the problem by adding/modifying some code lines.

```
...
int nbThreads_best;
int nbThreadsBlock_best;
int timer;
float cumulate_time;
float best_time = (float) INT_MAX;

for (int nbThreads = 32; nbThreads <= 65536; nbThreads *= 2) {
  for (int nbThreadsBlock = 32; nbThreadsBlock <= 512; nbThreadsBlock *= 2) {
    cumulate_time = 0;
```

```
    for (int trial = 0; trial < 10; trial++) {
      ...

      nbBlocks = nbThreads/nbThreadsBlock;

      CUT_SAFE_CALL(cutCreateTimer(&timer));
      CUT_SAFE_CALL(cutStartTimer(timer));

      MoveIncrEvalKernel<<<nbBlocks,nbThreadsBlock>>>(...);

      cudaThreadSynchronize();
      CUT_SAFE_CALL(cutStopTimer(timer));
      cumulate_time += cutGetTimerValue(timer);
      CUT_SAFE_CALL(cutDeleteTimer(timer));
      ...
    } // end loop for with int trial

    printf("nb Threads = %d\n", nbThreads);
    printf("nb Threads per block = %d\n", nbThreadsBlock);
    printf("%f %f\n", cumulate_time, best_time);

    if (cumulate_time < best_time) {
      nbThreads_best = nbThreads;
      nbThreadsBlock_best = nbThreadsBlock;
      best_timer = associated_timer;
    }

  }
 }
...
```

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

4. The CUDA code given below provides the kernel evaluation of $N * (N-1)/2$ solutions to a given problem which are obtained by a slight variation of an initial solution. Each solution is evaluated by a single thread. To perform this computation, each thread needs its own intermediate structure declared in local memory (C structure). However, when the values of the local variables M and N

are high, the amount of available local memory on GPU may not be sufficient. Therefore, instead of using the local memory, the global memory must be used to host and store the C structure. Provide an efficient transformation in which the accesses to the global memory (i.e. the C structure) are coalesced. For doing that, you have to modify only the code lines involving the C structure.

```
#define MAX(a,b) (((a)>=(b))?(a):(b))

__global__ void evaluateKernel(int* p,int * d, int* S, int* scores,
                               int* lookup_first, int* lookup_second) {

  int id = blockIdx.x * blockDim.x + threadIdx.x;

  if (id < N*(N-1)/2) {

    int first = lookup_first[id];
    int second = lookup_second[id];
    int solution[N];
    int temp;

    for (int i = 0; i < N; i++)
      solution[i] = S[i];

    temp = solution[first];
    solution[first] = solution[second];
    solution[second] = temp;

    int C[M*N]; // declared in local memory

    C[solution[0]] = p[solution[0]];

    for (int j=1; j<N; j++)
      C[solution[j]] = C[solution[j-1]] + p[solution[j]];

    for (int i=1; i<M; i++)
      C[i * N + solution[0]] = C[(i-1) * N + solution[0]] + p[i * N + solution[0]];

    for (int i=1; i<M; i++)
      for (int j=1; j<N; j++)
        C[i * N + solution[j]] = MAX(C[i * N + solution[j-1]],
                                     C[(i-1) * N + solution[j]]) + p[i * N + solution[j]];

    scores[id] = C[(M-1) * N + solution[N-1]];

  }

}
```

.....
.....
.....
.....

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Exercise 3 - GPU Computing using CUDA

The objective in this exercise is to find the minimal value of a given array containing a large number of elements. Write a complete program (i.e. host and device codes) which performs this operation on GPU in an optimized way. For the sake of simplicity, consider that the number of elements in the array is a power of 2.

```
void RandomInit(float* data, int n)
{
  for (int i = 0; i < n; ++i)
    data[i] = rand() / (float)RAND_MAX;
}

...
```

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....