Parallelizing a for loop (1D Naive Convolution) in CUDA

Asked 10 years ago Modified 9 years, 10 months ago Viewed 3k times

3

Can someone please help me convert a nested for loop into a CUDA kernel? Here is the function I am trying to convert into a CUDA kernel:

```
// Convolution on Host
void conv(int* A, int* B, int* out) {

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            out[i + j] += A[i] * B[j];
}
```

I have tried very hard to parallelize this code.
Here is my attempt:

```
__global__ void conv_Kernel(int* A, int* B, int* out) {

    int i = blockIdx.x;
    int j = threadIdx.x;

    __shared__ int temp[N];

    __syncthreads();
    temp[i + j] = A[i] * B[j];
    __syncthreads();

    int sum = 0;
    for (int k = 0; k < N; k++)
        sum += temp[k];
    out[i + j] = sum;
}
```

c cuda parallel-processing convolution

Share Improve this question Follow

asked Dec 2, 2014 at 0:22

user8919

67 ● 2 ● 9

1 Answer

Sorted by: Highest score (default)

6

Your cpu `conv` function appears to be doing this (for `N` = 4, as an example):

```
A0B0  A0B1  A0B2  A0B3                  +     ^
      A1B0  A1B1  A1B2  A1B3            +     N
            A2B0  A2B1  A2B2  A2B3      +    rows
                  A3B0  A3B1  A3B2  A3B3 =     v
-----------------------------------------
out0  out1  out2  out3  out4  out5  out6
    <-  (2*N)-1 columns ->
```

Your convolution is (to me) distinguished by the fact that it is convolving 2 signals of equal length. Since the GPU likes to work on "large" problems, this implies `N` should be large. However one immediate problem with your `conv_Kernel` realization is that it implies that the block dimension will be used to index into `A`, and the thread dimension will be used to index into `B`. But the thread dimension (`threadIdx.x`) is limited to 512 or 1024 for current CUDA GPUs. This will relegate us to only solving pretty small problems.

There are various other problems with your realization. One problem is that the shared memory size allocated is not enough to fit the `i+j` range (which can go from 0->2*(N-1)). This is trivial to fix of course, but the more serious issue is that I don't see a way to map your arithmetic onto anything resembling the desired pattern above. After spending a little while thinking about your kernel, I discarded it.

The convolution problem has a great deal of research associated with it, and can be optimized in various ways for massively parallel architectures like the GPU. Therefore I will focus on two very simple realizations which immediately suggest themselves based on the diagram above.

The first realization is simply to re-create the diagram above. We will create an intermediate `temp` array to store all the individual AxBy products, calculating and storing these products in the `conv_Kernel`. We will then launch a second kernel (`sum_Kernel`) which simply sums columns of the `temp` array, to produce the various `out` values. The first kernel requires `N` threads, which will successively calculate each row of the above diagram, in a slanting fashion as we iterate through `N` for-loop iterations, one per row. The second kernel requires (2*N)-1 threads, one for each column/`out` value.

My second realization (conv_Kernel2) dispenses with the need for a `temp` array, and just assigns one thread to each column/`out` value, and iterates through the `N` rows, computing the necessary products row-by-row, and summing those products "on-the-fly". The sum result is then directly stored in the `out` array.

Considering only the calculations, and not the time required for data movement/initialization, the GPU realizations begin to be faster than the naive single-

threaded CPU implementation at around `N`=512 on a K20x GPU, which is what I happened to be using. The second realization is also commended by the fact that the only data movement required is for A, B, and the result. The first realization requires in addition the `temp` array to be allocated and initialized to all zeroes. The size of the `temp` array is proportional to `N`*`N`, so the second realization also has the benefit that it does not require this temporary storage.

Here's a fully worked test case, running and timing the CPU realization you provided plus the two slightly different GPU realizations that I created:

```
$ cat t617.cu
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define N 4096
#define RG 10
#define USECPSEC 1000000ULL
#define nTPB 256


void conv(int* A, int* B, int* out) {

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            out[i + j] += A[i] * B[j];
}

unsigned long long dtime_usec(unsigned long long prev){
  timeval tv1;
  gettimeofday(&tv1,0);
  return ((tv1.tv_sec * USECPSEC)+tv1.tv_usec) - prev;
}


__global__ void conv_Kernel(int* A, int *B, int* temp) {

    int idx = threadIdx.x+blockDim.x*blockIdx.x;
    if (idx < N){
      int my_B = B[idx];
      for (int i = 0; i < N; i++)
        temp[idx + (i*2*N) + i] = my_B * A[i];
      }
}

__global__ void sum_Kernel(int *temp, int *out){
    int idx = threadIdx.x+blockDim.x*blockIdx.x;
    if (idx < (2*N)-1){
      int my_sum = 0;
      for (int i = 0; i < N; i++) my_sum += temp[idx + (i*2*N)];
      out[idx] = my_sum;}
}

__global__ void conv_Kernel2(int *A, int *B, int *out){
    int idx = threadIdx.x+blockDim.x*blockIdx.x;
    if (idx < (2*N)-1){
```

```
      int my_sum = 0;
      for (int i = 0; i < N; i++)
        if (((idx < N) && (i <= idx)) || ((idx >= N) && (i > (idx-N)))) my_sum
+= A[i]*B[idx-i];
      out[idx] = my_sum;
    }
}

int main(){

  int *h_A, *d_A, *h_result, *d_result, *result, *h_B, *d_B, *A, *B, *d_temp;

  B   = (int *)malloc(N*sizeof(int));
  A   = (int *)malloc(N*sizeof(int));
  h_A = (int *)malloc(N*sizeof(int));
  h_B = (int *)malloc(N*sizeof(int));
  h_result = (int *)malloc(2*N*sizeof(int));
  result   = (int *)malloc(2*N*sizeof(int));

  cudaMalloc(&d_B, N*sizeof(int));
  cudaMalloc(&d_A, N*sizeof(int));
  cudaMalloc(&d_result, 2*N*sizeof(int));
  cudaMalloc(&d_temp, 2*N*N*sizeof(int));

  for (int i=0; i < N; i++){
    A[i] = rand()%RG;
    B[i] = rand()%RG;
    h_A[i] = A[i];
    h_B[i] = B[i];}

  for (int i=0; i < 2*N; i++){
    result[i]   = 0;
    h_result[i] = 0;}

  unsigned long long cpu_time = dtime_usec(0);
  conv(A, B, result);
  cpu_time = dtime_usec(cpu_time);

  cudaMemcpy(d_A, h_A, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemset(d_result, 0, 2*N*sizeof(int));
  cudaMemset(d_temp, 0, 2*N*N*sizeof(int));

  unsigned long long gpu_time = dtime_usec(0);
  conv_Kernel<<<(N+nTPB-1)/nTPB,nTPB>>>(d_A, d_B, d_temp);
  sum_Kernel<<<((2*(N-1))+nTPB-1)/nTPB, nTPB>>>(d_temp, d_result);
  cudaDeviceSynchronize();
  gpu_time = dtime_usec(gpu_time);

  cudaMemcpy(h_result, d_result, 2*N*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < 2*N; i++) if (result[i] != h_result[i]) {printf("mismatch
at %d, cpu: %d, gpu %d\n", i, result[i], h_result[i]); return 1;}
  printf("Finished.  Results match.  cpu time: %ldus, gpu  time: %ldus\n",
cpu_time, gpu_time);


  cudaMemset(d_result, 0, 2*N*sizeof(int)); // just for error checking, the
kernel2 require no initialization of the result

  gpu_time = dtime_usec(0);
  conv_Kernel2<<<((2*(N-1))+nTPB-1)/nTPB,nTPB>>>(d_A, d_B, d_result);
  cudaDeviceSynchronize();
```

```
  gpu_time = dtime_usec(gpu_time);

  cudaMemcpy(h_result, d_result, 2*N*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < 2*N; i++) if (result[i] != h_result[i])
{printf("mismatch2 at %d, cpu: %d, gpu %d\n", i, result[i], h_result[i]);
return 1;}
  printf("Finished.  Results match.  cpu time: %ldus, gpu2 time: %ldus\n",
cpu_time, gpu_time);
  return 0;
}
$ nvcc -arch=sm_35 -o t617 t617.cu
$ ./t617
Finished.  Results match.  cpu time: 69059us, gpu  time: 3204us
Finished.  Results match.  cpu time: 69059us, gpu2 time: 1883us
$ nvcc -arch=sm_35 -O3 -o t617 t617.cu
$ ./t617
Finished.  Results match.  cpu time: 13750us, gpu  time: 3214us
Finished.  Results match.  cpu time: 13750us, gpu2 time: 1886us
$
```

(note that even just using the -O3 parameter makes a significant difference in the CPU code execution)

As I mentioned, I would consider both of my examples to be also quite "naive" for GPU code (niether uses shared memory, for example), but they may give you some ideas for how to get started.

For brevity of presentation, I have dispensed with CUDA error checking. However, I would suggest that any time you are having trouble with a CUDA code, that you perform proper cuda error checking. In the case of your `conv_Kernel`, I believe it would have indicated some errors (if you tried to run it.) As a quick test, you can always run any CUDA code with `cuda-memcheck` to see if any API errors are occurring.

EDIT: I tried a simple shared memory version of my `conv_Kernel2` but it wasn't any faster. I believe the reason for this is that these data sets (at `N`=4096, `A` and `B` are 16Kbytes each, `out` is approximately 32Kbytes) are small enough to easily fit in the GPU L2 cache, with no thrashing.

However, for newer architectures (cc 3.5 and newer) the CUDA compiler can sometimes make additional optimizations if the read-only input data is properly identified as such to the kernel. Therefore if we change my `conv_Kernel2` definition to:

```
__global__ void conv_Kernel2(const int * __restrict__ A, const int *
__restrict__ B, int *out){
```

then I witness slightly improved execution times, in my case:

```
$ ./t617
Finished.  Results match.  cpu time: 13792us, gpu  time: 3209us
```

```
Finished.  Results match.  cpu time: 13792us, gpu2 time: 1626us
$
```

I created a modified version of the code which does the following:

1. `N` is specified on the command line
2. only the cpu `conv` and gpu `conv_Kernel2` are included.
3. time cost to move the data to/from the GPU is included in the GPU timing measurement
4. a `typedef ... mytype;` is provided so that the code can be re-compiled easily to test behavior with various datatypes.
5. a "speedup factor" is printed out, which is the cpu time divided by the gpu time.

modified code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

// RG*RG*MAXN must fit within mytype

#define MAXN 100000
#define RG 10
#define USECPSEC 1000000ULL
#define nTPB 256

typedef double mytype;

void conv(const mytype *A, const mytype *B, mytype* out, int N) {

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            out[i + j] += A[i] * B[j];
}

unsigned long long dtime_usec(unsigned long long prev){
  timeval tv1;
  gettimeofday(&tv1,0);
  return ((tv1.tv_sec * USECPSEC)+tv1.tv_usec) - prev;
}



__global__ void conv_Kernel2(const mytype * __restrict__ A, const mytype *
__restrict__ B, mytype *out, const int N){
    int idx = threadIdx.x+blockDim.x*blockIdx.x;
    if (idx < (2*N)-1){
      mytype my_sum = 0;
      for (int i = 0; i < N; i++)
        if (((idx < N) && (i <= idx)) || ((idx >= N) && (i > (idx-N)))) my_sum
+= A[i]*B[idx-i];
      out[idx] = my_sum;
    }
```

```
}

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


  mytype *h_A, *d_A, *h_result, *d_result, *result, *h_B, *d_B, *A, *B;
  if (argc != 2) {printf("must specify N on the command line\n"); return 1;}
  int my_N = atoi(argv[1]);
  if ((my_N < 1) || (my_N > MAXN)) {printf("N out of range\n"); return 1;}
  B   = (mytype *)malloc(my_N*sizeof(mytype));
  A   = (mytype *)malloc(my_N*sizeof(mytype));
  h_A = (mytype *)malloc(my_N*sizeof(mytype));
  h_B = (mytype *)malloc(my_N*sizeof(mytype));
  h_result = (mytype *)malloc(2*my_N*sizeof(mytype));
  result   = (mytype *)malloc(2*my_N*sizeof(mytype));

  cudaMalloc(&d_B, my_N*sizeof(mytype));
  cudaMalloc(&d_A, my_N*sizeof(mytype));
  cudaMalloc(&d_result, 2*my_N*sizeof(mytype));

  for (int i=0; i < my_N; i++){
    A[i] = rand()%RG;
    B[i] = rand()%RG;
    h_A[i] = A[i];
    h_B[i] = B[i];}

  for (int i=0; i < 2*my_N; i++){
    result[i]   = 0;
    h_result[i] = 0;}

  unsigned long long cpu_time = dtime_usec(0);
  conv(A, B, result, my_N);
  cpu_time = dtime_usec(cpu_time);

  cudaMemset(d_result, 0, 2*my_N*sizeof(mytype));

  unsigned long long gpu_time = dtime_usec(0);
  cudaMemcpy(d_A, h_A, my_N*sizeof(mytype), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, my_N*sizeof(mytype), cudaMemcpyHostToDevice);
  conv_Kernel2<<<((2*(my_N-1))+nTPB-1)/nTPB,nTPB>>>(d_A, d_B, d_result, my_N);
  cudaDeviceSynchronize();
  cudaMemcpy(h_result, d_result, 2*my_N*sizeof(mytype),
cudaMemcpyDeviceToHost);
  gpu_time = dtime_usec(gpu_time);

  for (int i = 0; i < 2*my_N; i++) if (result[i] != h_result[i])
{printf("mismatch2 at %d, cpu: %d, gpu %d\n", i, result[i], h_result[i]);
return 1;}
  printf("Finished.  Results match.  cpu time: %ldus, gpu time: %ldus\n",
cpu_time, gpu_time);
  printf("cpu/gpu = %f\n", cpu_time/(float)gpu_time);
  return 0;
}
```

Share

Improve this answer

Follow

edited May 23, 2017 at 12:13

Community Bot

1 • 1

answered Dec 2, 2014 at 4:40

Robert Crovella

151k • 12 • 243 • 290