C2CU: A CUDA C Program Generator for Bulk Execution

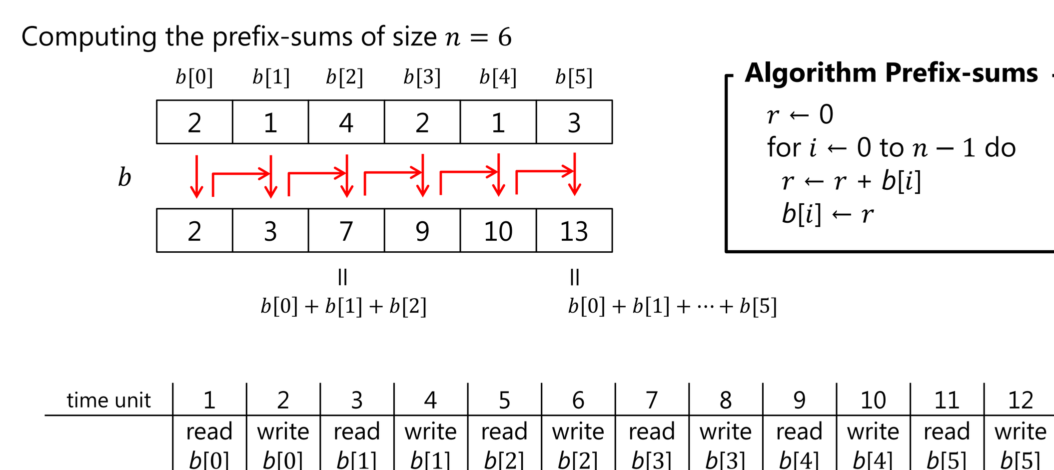
Yasuaki Ito
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527, JAPAN

Abstract

We present a time-optimal implementation for bulk execution of an oblivious sequential algorithm. Our second contribution is to develop a tool, named C2CU, which automatically generates a CUDA C program for a bulk execution of an oblivious sequential algorithm.

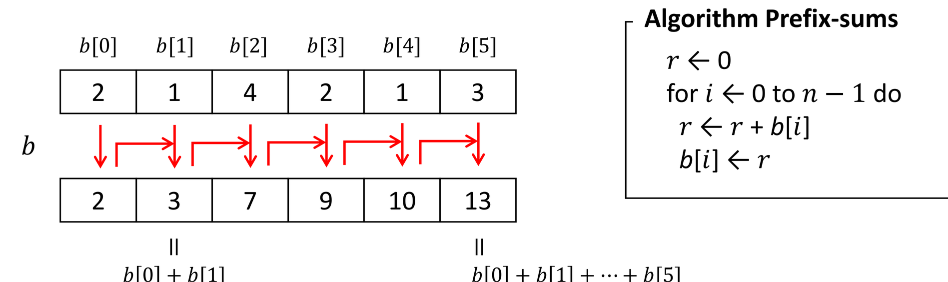
Oblivious algorithm

A sequential algorithm is oblivious if an address accessed at each time unit is independent of the input.

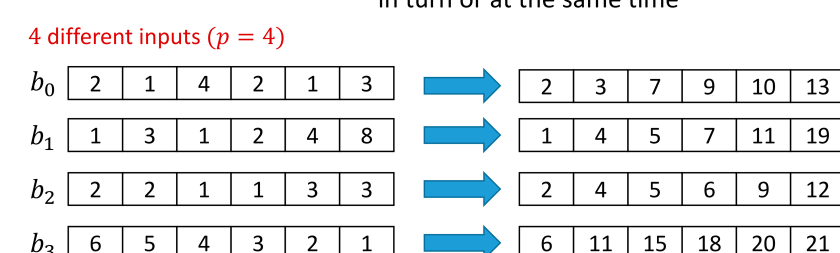


Bulk execution of oblivious algorithms

Computing the prefix-sums of size $n = 6$

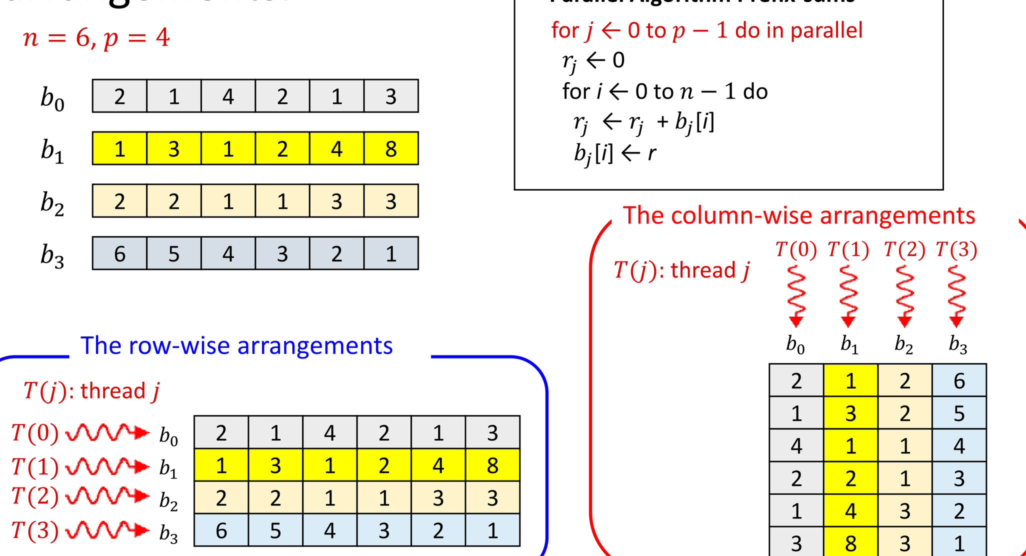


The bulk execution of prefix-sums: compute the prefix-sums for many different inputs in turn or at the same time



Bulk execution with coalesced access

Every memory access can be performed by coalesced access using column-wise arrangements.



C2CU: A CUDA C Program Generator for Bulk Execution

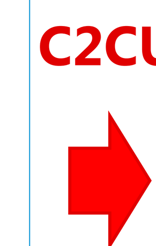
C2CU automatically **converts a C language program** describing an oblivious sequential algorithm into **a CUDA C program** that performs the bulk execution of the C language program.

An example of conversion for Floyd-Warshall algorithm

```
 1: #define N 1024
 2: float D[N][N];
 3: void update_dist(int i, int j, int k){
 4:   if( D[i][j] > D[i][k] + D[k][j] ) {
 5:     D[i][j] = D[i][k] + D[k][j];
 6:   }
 7: }
 8:
 9: void floyd_warshall(){
10:   int i,j,k;
11:   for(k=0;k<N;k++) {
12:     for(i=0;i<N;i++) {
13:       for(j=0;j<N;j++) {
14:         update_dist(i,j,k);
15:       }
16:     }
17:   }
18: }
19:
20: int main(int argc, char *argv[]){
21:   input_array();
22: #pragma kernel
23:   floyd_warshall();
24:   ...
```

Sequential C program

C2CU



```
 1: #define N 1024
 2: #define __P__ 2048
 3: #define __T__ 64
 4: #define __B__  __P__/__T__
 5: float D[N][N][__P__];
 6: __device__ float __D[N][N][__P__];
 7:
 8: __device__ void update_dist(int i, int j, int k, int __id__){
 9:   if( __D[i][j][__id__] > __D[i][k][__id__] + __D[k][j][__id__] ) {
10:     __D[i][j][__id__] = __D[i][k][__id__] + __D[k][j][__id__];
11:   }
12: }
13:
14: __global__ void floyd_warshall(){
15:   int __id__ = blockIdx.x * blockDim.x + threadIdx.x;
16:   int i,j,k;
17:   for(k=0;k<N;k++) {
18:     for(i=0;i<N;i++) {
19:       for(j=0;j<N;j++) {
20:         update_dist(i,j,k,__id__);
21:       }
22:     }
23:   }
24: }
25:
26: int main(int argc, char *argv[])
27: {
28:   input_array();
29: #pragma kernel
30:   cudaMemcpyToSymbol(__D, D, sizeof(float)*N*N*__P__, 0);
31:   floyd_warshall<<<__B__,__T__>>>();
32:   cudaMemcpyFromSymbol(D, __D, sizeof(float)*N*N*__P__, 0);
33:   ...
```

Generated CUDA C program

Performance Evaluation

Bulk execution of oblivious algorithms:

- Floyd-Warshall algorithm
- Bitonic sorting algorithm, and
- Montgomery modulo multiplication.

GPU: NVIDIA GeForce GTX TITAN using p threads in $\frac{p}{64}$ CUDA blocks with 64 threads each

CPU: Intel Core i7 (3.5MHz)
The sequential algorithm is repeated by p times.

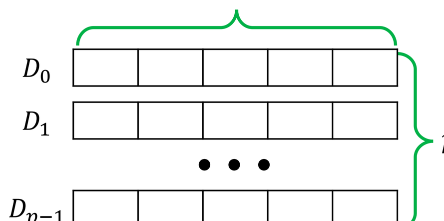
Our implementations running on GeForce GTX Titan for the bulk execution can be
54 times faster for Floyd-Warshall algorithm,
199 times faster for bitonic sort, and
78 times faster for Montgomery modulo multiplication,
over the implementations on a single CPU.

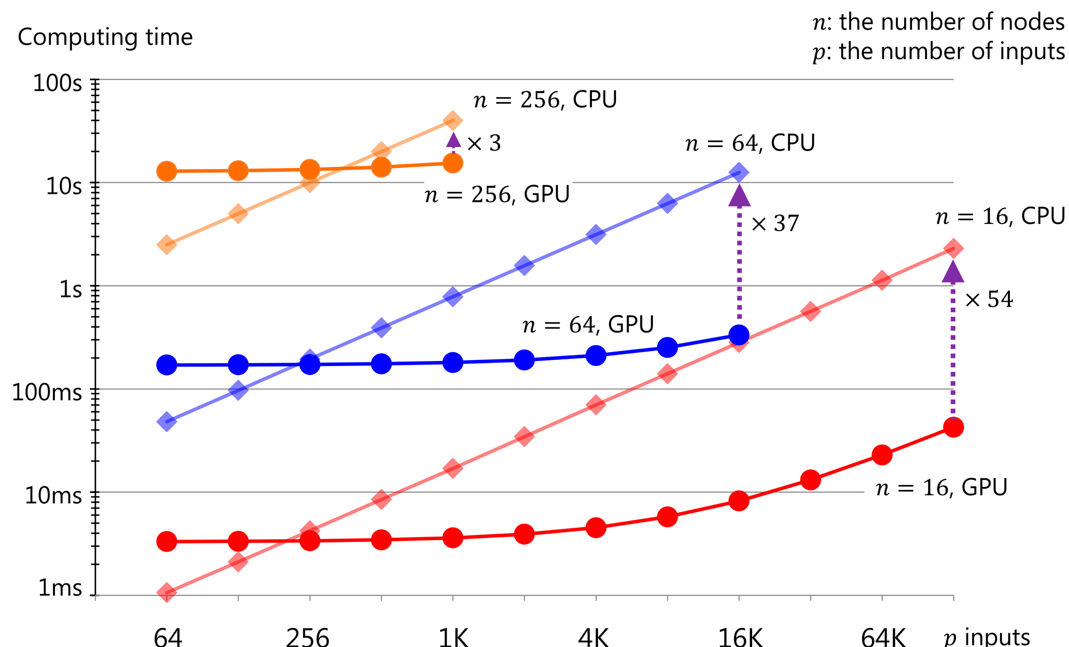
Floyd-Warshall algorithm

Execute Floyd-Warshall Algorithm for p inputs

Floyd-Warshall Algorithm:
Compute the distance of the shortest paths of all pairs of nodes in a directed graph

Instance:
$n = 16, 64, 256$
(directed graphs with 16 nodes, 64 nodes, and 256 nodes)
$p = 64, 128, 256, \ldots, 128K(=131072)$

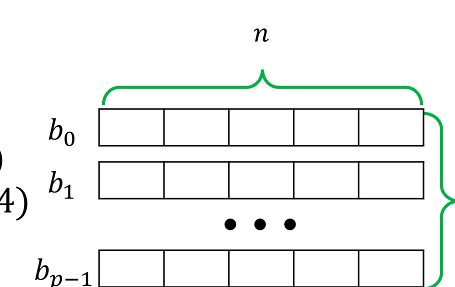


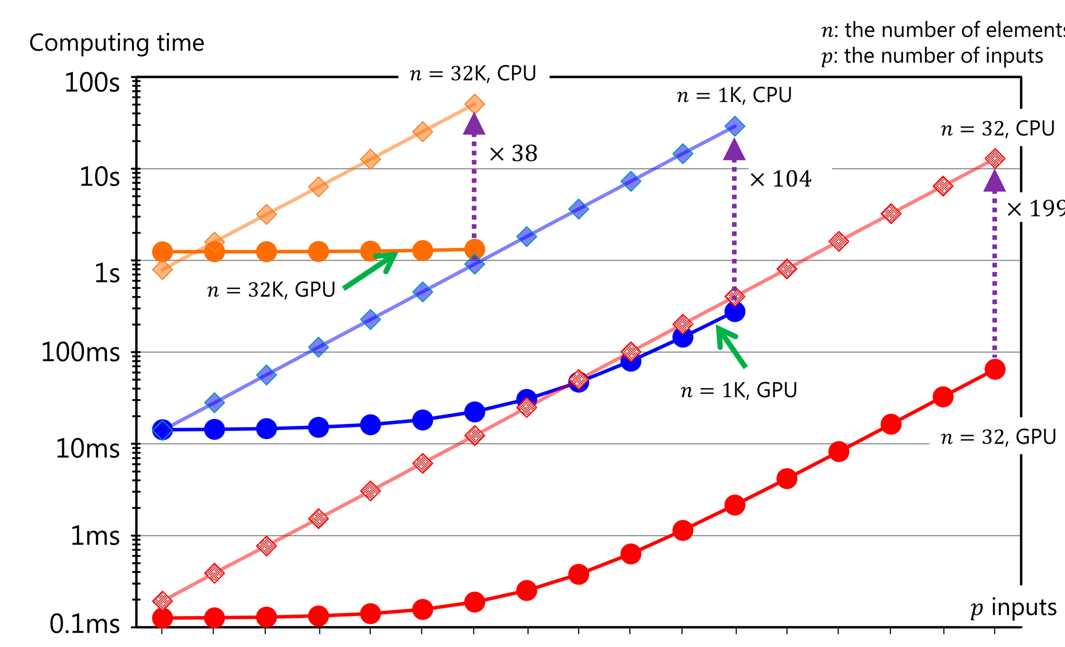


Bitonic sort

Execute Bitonic sort for p inputs

Instance:
$n = 32, 1K(=1024), 32K(=32768)$
$p = 64, 128, 256, \ldots, 4M (=4194304)$
input data type: float





Montgomery modulo multiplication

Execute Montgomery modulo multiplication for p inputs

Input:
$n = 512, 16K(=16384), 1M(=1048576)$
$p = 64, 128, 256, \ldots, 2M (=2097152)$