Bull. Nov. Comp. Center, Comp. Science, 39 (2016), 51–60

Implementation of the STAR-machine on GPU

T.V. Snytnikova, A.V. Snytnikov

Abstract. In this paper, we present the simulation of an abstract model of SIMD type with vertical data processing (the STAR-machine) on GPU with CUDA framework. There is a number of algorithms developed for the STAR-machine. The research conducted recently shows that such a model is extremely efficient when used to solve graph problems. Associative operations are the key properties of this model. In particular, all of them take constant time. In this paper, we present an implementation of associative operations on GPU (Graphic Processing Units). This study is aimed at providing a bridge or a general manual instruction to convert the STAR algorithms to the GPU implementation. As the architecture of the STAR-machine in modern technologies has not been built yet, this provides a possible way to implement the STAR algorithms on an alternative platform to verify their correctness and efficiency, especially, for massive data input.

1. Introduction

The associative (content addressable) parallel processors of the SIMD type with vertical processing and simple single-bit processing elements ideally suit to performing fast parallel search operations being used in different applications such as graph theory, relational database processing, image processing. Such an architecture performs data parallelism at the basic level, provides a massively parallel search by contents, and allows using two-dimensional tables as basic data structures [1]. This class of parallel computers includes the well-known systems STARAN, DAP, MPP, and CM-2. The STAR-machine is an abstract model of the SIMD type. There have been a number of algorithms [2–10] developed with this model.

While a great number of the STAR algorithms are being developed, their implementation becomes an interesting problem. Because of the platform STARAN is not accessible in todays computer lab settings, it is impossible to truly evaluate the performance of a STAR algorithm with massive data input. The system VisualStar has been developed to simulate the STAR-machine [11]. It allows one to edit, to compile, and to implement procedures written in the STAR language. The VisualStar is useful to develop and to test algorithms, but not to calculate a a great body of data because of sequential computation.

We look for a platform that is able to implement the STAR-machine so as to run its algorithms in massively parallel model with high efficiency. An ideal platform must have SIMD architecture and also be easily accessible. There is no doubt that Nvidia GPUs are an excellent choice.

The main objective of using GPU is to accelerate intensive graphic data processing. Later, with introduction of Nvidia CUDA (compute unified device architecture), a high-level programming interface, GPU was evolved to be a powerful computing platform to support the general purpose parallel computation. It has been used in numerous application fields for massively parallel data processing [12]. The GPU is a typical SIMD architecture and is especially good for fine-grained large amount data-intensive parallel computation. Its features provide the possibility of implementing associative parallel algorithms with easy accessibility and high scalability.

To implement a STAR algorithm on an architecture, we need to find a way to execute each basic operation of the language STAR in the corresponding running environment. In this paper, this is our contribution.

A similar work is performed for the other associative computing model (MASC) [13].

This paper is organized as follows. The model of the STAR-machine is described in Section 2. Section 3 presents the implementation steps for each basis operation of the STAR-mashine on GPU. Section 4 gives the performance analysis.

2. The STAR-machine model

The STAR-machine is a model of the SIMD type with vertical data processing. It consists of the following components:

- a sequential control unit (CU), where programs and scalar constants are stored;
- an associative processing unit consisting of p single-bit processing elements (the PEs);
- a matrix memory for the associative processing unit.

The CU passes an instruction to all PEs in one unit of time. All active PEs execute it in parallel, while inactive PEs do not perform it. Activation of a PE depends on the data. It should be noted that the time of performing any instruction does not depend on the number of processing elements [14].

To input binary data are loaded into the matrix memory in the form of two-dimensional tables in which each datum occupies an individual row and it is updated by a dedicated processing element. It is assumed that there are a greater number of PEs than data. The rows are numbered from top to bottom and the columns – from left to right. Both a row and a column can be easily accessed. Some tables may be loaded into the matrix memory.

An associative processing unit is represented as h vertical registers ($h \geq 4$), each consisting of p bits. The vertical registers can be regarded as a one-column array. The bit columns of the tabular data are stored in

the registers which perform necessary Boolean operations and record the search results. The STAR-machine run is described by means of the language STAR [15] which is an extension of Pascal. Let us briefly consider the STAR basis constructions. To simulate data processing in the matrix memory, we use data types **word**, **slice**, and **table**. Constants for the types **slice** and **word** are represented as a sequence of symbols of $\{0, 1\}$ enclosed within single quotation marks. The types **slice** and **word** are used for the bit column access and the bit row access, respectively, and the type **table** is used for defining the tabular data. Assume that any variable of the type **slice** consists of p components which belong to $\{0, 1\}$. For simplicity, let us call **slice** any variable of the type **slice**.

Now, we present some elementary operations and predicates for **slices**.

Let X, Y be variables of the type **slice** and i be a variable of the type **integer**. We use the following operations:

SET(Y) sets all the components of Y to $'1'$;

CLR(Y) sets all the components of Y to $'0'$;

$Y(i)$ selects the ith component of Y;

FND(Y) returns the ordinal number i of the first (or the uppermost) component $'1'$ of Y, $i \geq 0$;

STEP(Y) returns the same result as FND(Y) and then resets the first component $'1'$ to $'0'$.

In the usual way we introduce the predicates ZERO(Y) and SOME(Y) and the bitwise Boolean operations X *and* Y, X *or* Y, *not* Y, and X *xor* Y.

Let T be a variable of the type **table**. We employ the following two operations:

ROW(i, T) returns the ith row of the matrix T;

COL(i, T) returns the ith column of T.

Remark 1. All operations for the type **slice** can also be performed for the type **word**.

Remark 2. Note that the STAR statements [2] are defined in the same manner as for Pascal.

3. Associative operations on GPU

In this section, implementation steps for the basis operations of the STAR-machine on the GPU platform are presented. First, the realization of data types is described. Then each associative operation is discussed one by one in regard of its implementation on GPU.

3.1. The types of data. As noted above, to simulate data processing in the matrix memory, we use the data types **word**, **slice**, and **table** in the STAR language.

The GPU uses a conventional way to store data (Random Access Memory). A data item is identified by its memory address. The CUDA C is an extension of C language. Thus, to simulate the data types, we declare classes of the same name. Class **Slice** is used to simulate the type **word**, too.

```
class Slice{
    // Host memory:
    bool word_flag;
    // h_v[N] is used for input/output of a slice
    Unsigned long long int h_v[N];

    // Device memory:
    // *d_v stores the address of first element of the array
    // of the size N
    Unsigned long long int *d_v;
...}
```

Here N is equal to the length of **slice** divided into 64 (the size of long long int type) rounded up. The array `h_v[N]` is used for input/output of a slice. It is stored in the host memory, and the device pointer `*d_v` stores the address of the first element of the array of the size N of unsigned long long in the device global memory. All calculations are performed with the array `d_v`.

The class Table consists of the array of M objects of the class Slice and some pointers.

```
class Table{
    Slice table[M];
    LongPointer *slice_device_pointer_table;
    // Device memory:
    LongPointer *d_table, *d_slice_device_pointer_table;
...}
```

The device pointer `*d_table` stores the address of the first element of the array of $M \times N$ size of unsigned long long in the device global memory. The host pointer `*slice_device_pointer_table` and the device pointer `*d_slice_device_pointer_table` store the address of the first element of the array of N pointers to the columns of `d_table`.

3.2. The operations on a slice. The basis operations on the STAR type **slice** are realized as methods of the class Slice with the same name. And each method calls a global function, which performs the operation.

The operation X *and* Y is implemented as

```
Slice Slice::operator & (const Slice & b)
{
    and_long_values<<<N,1>>>(d_v, b.d_v);
    return *this;
}

__global__ void and_long_values(unsigned long long int *d_v,
unsigned long long int *d_v1)
{
    d_v[blockIdx.x] &= d_v1[blockIdx.x];
}
```

The operations SET(X), CLR(X) and the other bitwise Boolean operations are implemented in the same way. Then, performing these operations takes a longer time by a constant, and does not depend on the slice length.

Now, let us discuss the implementation of the operation FND(X). It is made in two stages:

- At the first stage, the array `d_v[N]` is reduced to one variable of the type unsigned long long. It performs a call of the global procedure `find`. At each level, it reduces the array X to the array new_x, whose size is 64 times smaller. Each non-zero element of X maps the bit $'1'$ of the new_x, and each zero element of X maps the bit $'0'$.
- At the second stage, the global procedure `first_backward` computes the result by expending up:

```
__global__ void first_backward(LongPointer *d_v,
    int *d_first_non_zero, int level)
{
    int f[LEVELS], iu;
    unsigned long long int *dvl, u;
    char lprt[100];

    f[level+1] = 1;
    while(level >= 0)
    {
        dvl = d_v[level];
        int index = f[level+1] - 1;
        u = dvl[index];

        f[level] = __ffsll(u) + index * SIZE_OF_LONG_INT;
        level--;
```

```
    }
    *d_first_non_zero = f[0];
}
```

Performing the operation FND(X) takes a longer time by the factor $O(\log N)$ for a slice with the length of N. The operation STEP(X), the predicates SOME(X) and ZERO(X) are performed in the same way.

3.3. The operators on the class Table. The operation COL(i, T) is performed by calling `T.col(i)`:

```
Slice *col(int i) { return &(table[i-1]); }
```

It is used to access to columns both for reading and writing.

The ROW(i, T) is implemented with two procedures. The procedure `SetRow(Slice *s, int i)` is used to write the **word** s into the ith row of the table. And the function `Slice *Table::row(int i)` returns the pointer to the **word**, which matches with the ith row of the table. The same algorithm is used in both cases:

- In each column, there is an element, which includes one bit of the row. Thus, the following indices are computed: the element number and the bit number. The indices are the same for all columns.
- Bits are read or written in parallel by columns. We have M (the size of the table) variables `tmp` of the type `unsigned long long int`, in which the kth bit is equal to the ith bit of the kth column, and the other bits are equal to zero.
- To read the row, all bits are gathered into the **word** in the following manner.

```
d_v[ni] = get_array(tmp,0,M)
        | get_array(tmp,1,M)
        ...
        | get_array(tmp,63,M)
```

Performing the operations COL(i, T) and ROW(i, T) takes a longer time by a constant factor, but the COL(i, T) is faster than the ROW(i, T).

4. The performance analysis

Now let us consider the simulation of the Warshall associative parallel algorithm. This algorithm was described and proven in [2]. The procedure receives as input data an adjacency matrix of the graph with n vertices and returns the path matrix for the transitive closure of the graph.

```
proc WARSHALL(n: integer; var P: table);
/*Here n is the number of graph vertices.*/
    var X: slice(P); v,w: word; i,k: integer;
begin for k:=1 to n do
   begin
      x := COL(k,P);
      w := ROW(k,P);
      while SOME(X) do
      begin
        i := STEP(X);
        v := ROW(i,P);
        v := v or w;
        ROW(i,P) := v;
      end;
   end;
end;
```

It uses the following basic operations: or, SOME, COL (to read), ROW (to read and to write), and STEP. It is sufficient to consider these operations, because other operations are realized and run by the same way.

4.1. The basis operations runtime. To estimate the running time, the Warshall algorithm is performed on the two graph examples: with 100 and 1000 vertices. The result of profiling is shown in the table.

The basic operation ROW(i, T) is simulated by the procedures `get_row` and `set_row`. The first of them runs a similar time, which does not depend on the size of a row. The run time of the second procedure grows twice by increasing the size of a row by a factor of 10.

The procedures `find` and `first_backward` perform the operation FND(X) (one calls the procedure `put` after them to implement STEP(X)). Their run time is similar because reducing the arrays is performed once for the number of vertices smaller than $64 \times 64 = 4096$. The number of reduc-

Profiling of the basic operations

Procedure	100 vertices			1000 vertices		
	Calls	Time (ms)	Avg (µs)	Calls	Time (s)	Avg (µs)
get_row	7587	141.95	18.71	524427	9.70	18.49
set_row	7487	31.16	4.16	523870	5.98	11.23
find	15174	34.25	2.26	1048855	3.06	3.37
first_backward	15174	51.23	3.38	1048855	3.54	3.37
put	7487	21.95	2.93	523868	1.54	2.94
or_long_value	7487	13.97	2.18	523870	1.01	1.93

tion operations and the number of vertices are related in the following way: n reducing for the number of vertices in $[64^n, 64^{n+1}]$.

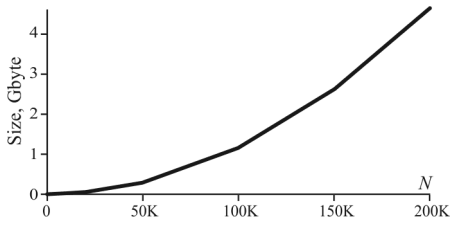
The procedure `put` is used to implement the basic operations $X(i)$ and STEP(i). It runs in a constant time.

The procedure `or_long_value` performs the basic operation $X\,or\,Y$ and runs in parallel at a constant time. The other bitwise Boolean operations are similarly situated.

The basic operation COL(i, T) is performed by passing a pointer to the column.

4.2. The data size. Now we consider the issues associated with the data size.

If a graph with n vertices is represented as an adjacency matrix, then we need $8(n(\lceil n/64\rceil + 2) + 1)$ bytes to keep it. The correlation between the size of the global memory and the number of graph vertices is shown on the figure. Thus, we need ≈ 3.78 Gbyte of the global memory for a graph with 180K vertices.



The correlation between the number of graph vertices and the size of the global memory

If an unweighted graph with n vertices and m edges is represented as a list of edges and a list of nodes, we need $2(8(\log_2 n \cdot (\lceil m/64\rceil + 2) + 1)) + 8(\log_2 n \cdot (\lceil n/64\rceil + 2) + 1)$ bytes to keep it. In this case, it is necessary to have about 2.39 Mbyte to keep a graph with 10K vertices and 10M edges. For example, the graph of MemeTracker phrases and hyperlinks between 96 million blog posts from Aug 2008 to Apr 2009 [19] has 96M vertices and 418M edges and needs about 2.63 Gbyte to keep in the global memory. For example, Kepler K40 GPU accelerator has 12 Gbyte memory and 2,880 thread processors.

4.3. Some optimization of the Warhall procedure. Note, that the procedure WARSHALL can be changed to decrease the runtime in view of the implementation. The procedure WARSHALL-1 receives input data as transposed adjacency matrix of graph with n vertices.

```
proc WARSHALL-1(n:integer; var P: table);
/* Here n is the number of graph vertices.
   P is transposed adjacency matrix. */
var X: word; v,w: Slice; i,k: integer;
begin for k:=1 to n do
   begin
      x := ROW(k,P);
```

```
      w := COL(k,P);
      i := STEP(X);
      while i>0 do
      begin
        v := COL(i,P);
        v := v or w;
        COL(i,P) := v;
        i := STEP(X);
      end;
   end;
end;
```

5. Conclusion

In this paper, we have proposed the implementation of associative operations of the STAR-machine on the GPU architecture. As is shown in Section 4, most of these associative operations can be implemented on GPU with an extra $O(\log N)$ efficiency loss in theory. However, as is shown in Section 5, the associative operators are implemented with an efficiency close to constant. This is due to the fact that software programmed steps on GPU can be directly used to convert a STAR algorithm so as to be implemented on the GPU-CUDA platform.

In the future, we are planing to implement of the library of basis and auxiliary procedures from [2, 3]. And, perhaps, some blocks of the STAR operators can be performed taking into account the differences between the STAR-machine and GPU.

Our purpose is to extend the implementation to a technology, which would allow one to use and to develop associative parallel algorithms.

References

[1] Potter J.L. Associative Computing: a Programming Paradigm for Massively Parallel Computers / Kent State University. — New York, London: Plenum Press, 1992.

[2] Nepomniaschaya A.S. Solution of path problems using associative parallel processors // Parallel and Distributed Systems. — 1997. — P. 610–617.

[3] Nepomniaschaya A.S., Dvoskina M.A. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors // Fundamenta Informaticae. — Amsterdam: IOS Press, 2000. — Vol. 43. — P. 227–243.

[4] Nepomniaschaya A.S. An associative version of the Bellman–Ford algorithm for finding the shortest paths in directed graphs // Proc. 6th Int. Conf. PaCT-2001. — Springer, 2001. — P. 285–292. — (Lect. Notes in Comp. Sci.; 2127).

[5] Nepomniaschaya A.S. Efficient implementation of Edmonds' algorithm for finding optimum branchings on associative parallel processors // Proc. 8th Int. Conf. on Parallel and Distributed Systems (ICPADS'01), KyongJu City, Korea. — IEEE Computer Society Press, 2001. — P. 3–8.

[6] Nepomniaschaya A.S. Comparison of performing the Prim–Dijkstra algorithm and the Kruskal algorithm by means of associative parallel processors // Cybernetics and System Analysis. — 2000. — No. 2. — P. 19–27 (In Russian).

[7] Nepomniaschaya A.S. Representation of the Gabow algorithm for finding smallest spanning trees with a degree constraint on associative parallel processors // Proc. Euro-Par'96 Parallel Processing. Second Int. Euro-Par Conf. Lyon, France. — Springer, 1996. — P. 813–817. — (Lect. Notes in Comp. Sci.; 1123).

[8] Nepomniaschaya A.S., Borets T.V. Associative parallel algorithm of checking spanning trees for optimality // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosibirsk, 2002. — Iss. 17. — P. 75–88.

[9] Borets T.V. Associative parallel algorithm performing depth-first search // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosibirsk, 2003. — Iss. 19. — P. 15–24.

[10] Borets T.V. An associative version of the Lengauer–Tarjan algorithm for finding immediate dominators in a graph // Optoelectronics, Instrumentation and Data Processing. — 2003. — No. 3. — P. 21–28 (In Russian).

[11] Borets T.V. A programming system VisualStar // Proc. Conf. of Young Scientists / G.A. Michailov, ed. — Novosibirsk, 2004. — P. 20–26.

[12] Park I., Signhal N., Lee M., et al. Design and performance evaluation of image processing algorithms on GPUs // IEEE Transactions on Parallel and Distributed Systems. — January, 2011. — Vol. 22, No. 1. — P. 91–104.

[13] Jin M. Associative Operations from MASC to GPU // PDPTA'15 — 21st Int. Conf. on Parallel and Distributed Processing Techniques and Applications. — 2015. — P. 388–393.

[14] Foster C.C. Content Addressable Parallel Processors. — New York: Van Nostrand Reinhold Company, 1976.

[15] Nepomniaschaya A.S. Language STAR for associative and Parallel computation with vertical data processing // Proc. Int. Conf. "Parallel Computing Technologies". — Singapure: World Scientific, 1991. — P. 258–265.

[16] Kirk D.B., Hwu W.W. Programming Massively Parallel Processors. — Morgan Kaufmann Publishers, 2010.

[17] Harish P., Narayanan P.J. Accelerating large graph algorithms on the GPU using CUDA // IEEE High Performance Computing. — 2007. — P. 197–208.

[18] Nvidia CUDA. — http://www.nvidia.com/cuda/.

[19] Stanford Large Network Dataset Collection. — https://snap.stanford.edu/data/.