THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 20, Number 4/2019, pp. 411–417

ALL-PAIR SHORTEST PATH ON A HYBRID MAP-REDUCE BASED ARCHITECTURE

Voichița DRAGOMIR, Gheorghe M. ȘTEFAN

"Politehnica" University of Bucharest, ETTI
"Politehnica" University of Bucharest, ETTI
Corresponding author: VOICHITA DRAGOMIR, E-mail: `voichita.dragomir@upb.ro`

Abstract. All-Path Shortest-Path Problem (APSP) algorithm implemented on two currently used architectures, a mono-core one and on a many-core one, is compared with the implementation on Map-Reduce Architecture (MRA), a novel many-core architecture we propose. The hybrid system using an accelerator based on MRA is described. The system is evaluated in two versions for running the Floyd-Warshall APSP algorithm. A first version is for an accelerator with the number of cores, p, exceeding the number of vertexes, $|V|$, while the second uses a fix number of cells, N, for $|V| = N\times M$, where M is a power of 2. The programs, written for our accelerator running in simulation, are used to evaluate the execution time for both versions. The performance of a mono-core and of a GPU is compared with our MRA acceleration. A 128-cell MRA engine accelerates $118\times$ at $20\times$ less energy a mono-core, and $3\times$ at $26\times$ less energy a 128-core GPU.

Key words: APSP, parallelism, hybrid computation, map-reduce architecture, Floyd-Warshall algorithm.

1. INTRODUCTION

In [1], the problem of APSP is investigated from a theoretical point of view and the result is compared with other solutions in the same order of complexity. The APSP modified matrix-multiplication based algorithm takes as input the matrix:

$$\mathbf{D} = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1p} \\ d_{21} & d_{22} & \dots & d_{2p} \\ \vdots & & & \\ d_{p1} & d_{p2} & \dots & d_{pp} \end{bmatrix}$$

where d_{ij} is the distance between the vertex i and the vertex j, for $i,j = 1,2,\dots p$. On a p-processor hypercube architecture the algorithm is evaluated as working in $O(N^2 \log N)$ cycles [5], where N is the number of vertices and $p \geq N$. Our architecture presented in [1] provides the same theoretical time performance, but has the advantages of the engine size which is $O(p)$, rather than the hypercube with the size in $O(p \log p)$. So, for small N, because the coefficient of N^2 is small[1], the modified matrix-multiplication based algorithm for APSP works well enough. But, for big N the Floyd-Warshall algorithm must be considered. Its three embedded loops:

```
/*****************************************
Floyd-Warshall algorithm on scalars
*****************************************/
function FW(N)
 for (k=0; k<N; k=k+1)
  for (k=i ; i<N; i=i+1)
   for (j=0; j<N; j=j+1)
    d[ij] <= min(d[ij], (d[ik] + d[kj]))
```

[1] The execution time reported in [1] is improved, due to architectural improvements, to $2N^2+44N+0{:}5\log_2 P+2$.

are executed in time belonging to $O(N^3)$. Using an N-cell parallel architecture we expect an acceleration in $O(N)$. But, the actual constants are also important.

The next section is about the state of the art in using GPUs as accelerators for solving the APSP problem. The third section describes the architecture of the proposed accelerator. The fourth section evaluate the performance of the proposed accelerator in solving the APSP problem and compares it with the existing architectural solution. The last section concludes the paper.

2. STATE OF THE ART

Besides the structural performance, expressed as the acceleration

$$\alpha_{accelerated}^{reference} = \frac{t_{reference}(f_{clock}^{r})}{t_{accelerated}(f_{clock}^{a})}$$

the architectural acceleration

$$A_{accelerated}^{reference} = \alpha_{accelerated}^{reference} \times \frac{f_{clock}^{r}}{f_{clock}^{a}}$$

will be provided. The architectural acceleration is the acceleration evaluated with the two engines working at the same frequency. In the following, α results from measurements, while A is computed by scaling the performance of the fastest device to the frequency of the other.

Using data published in [2] and [3] the structural and the architectural acceleration provided for Floyd-Warshall algorithm by a GPU used as GPGPU are presented in Fig. 1. In the following, we will consider $t_{scalar} = 4.1 \times t_{SSE}$ CPU according to [2].

Nodes	t_{scalar}	t_{gpu}	α_{gpu}^{scalar}	A_{gpu}^{scalar}
64	0.00086 s	0.0014 s	0.61	1.65
128	0.00615 s	0.0025 s	2.46	6.66
256	0.04018 s	0.0077 s	5.20	14.10
512	0.21320 s	0.0349 s	6.08	16.47
1024	1.71790 s	0.2301 s	7.46	20.21
2048	14.39100 s	1.7356 s	8.29	22.46
4096	122.87700 s	13.7200 s	9.38	25.41

Fig. 1 – Structural and architectural performance for solving APSP problem with Floyd-Warshall algorithm on a mono-core scalar engine (3.66 GHz Pentium 4 without SSE, powered at 100 Watt) compared with GPU (1.35 GHz, 128-core Nvidia Quadro FX5600 powered at 178 Watt).

The structural acceleration of a GPU used as GPGPU remains under $10\times$ with an almost double power consumption, while the architectural performance remains under 25% of its peak performance at a more than $4\times$ higher energy use. GPU provides acceleration, $< 32\times$ with 128 processing units, but it is too small and the price paid in energy consumption is too high.

The reason for this low use of the high peak performance of a GPU consists in fatal architectural inadequacies: the use of a highly optimized GPU as general-purpose accelerator.

3. MAP-REDUCE ACCELERATOR

The accelerator with MRA is designed to be attached to any standard general-purpose processing unit to configure a hybrid computing system. Usually, the accelerator runs a (standard) library of computationally intense functions. The kernel of the accelerator consists of:

- CONTROLLER which issue in each cycle instruction (& data) to ARRAY
- linear ARRAY of cells (the MAP section) receiving from CONTROLLER instruction & data and sending back, through
- a REDUCTION network, scalars computed starting from vectors distributed along the cells.

The system is programmed using an instruction set which is the Cartesian product of two ISA, one for CONTROLLER (cISA) and another for ARRAY (ISA). The code line contains always two instructions:

```
cCCC; AAA;
```

where: `cCCC` is the instruction `CCC` for CONTROLLER, and `AAA` is the instruction to be executed is each active cell of ARRAY. Some instructions are labelled:

```
LB(label_name); cCCC; AAA;
```

An example of an actual program is presented in Figure 2.

The instruction set architecture in both, CONTROLLER and in ARRAY, is accumulator-based: the scalar register `acc` in CONTROLLER and the vector register `[acc[0], acc[1], ..., acc[p-1]]` in ARRAY.

The local memory resources are: the scalar memory `scalarMem[0:m-1]` for CONTROLLER, and the vector memory `vectorMem[0:p-1][0:m-1]` in ARRAY. Therefore, if *BBB* stands for a binary operation, then:

- `cBBB(X): acc <= acc BBB scalarMem[X]`
- `BBB(X): acc[i] <= acc[i] BBB vectorMem[i][X]`, in each active cell

Besides the standard arithmetic and logic operations defined for both, CONTROLLER and ARRAY, and the sequential control operations defined for CONTROLLER, there are also a few *spatial control* operations defined for ARRAY, as follows:

- `ACTIVATE`: all cells from the MAP section of ARRAY are activated
- `WHERE(cond)`: in all active cells the condition cond is tested and only the cells where it is fulfilled remain active
- `ENDWHERE`: the cells inactivated by the last `WHERE` or `ELSEWHERE` are reactivated
- `ELSEWHERE`: acts like the sequence: `ENDWHERE`; `WHERE(!cond);`

These spatial control instructions allow predicated vector execution in ARRAY. The system also allows embedded WHERE(...) operations.

Another specific instructions are the reduction instructions used to send, with a latency in $O(\log p)$, as co-operand for `acc` in CONTROLLER, the value computed by the REDUCTION network, as follows:

- the sum of `acc[i]` from all active cells
- _the maximum value of `acc[i]` from all active cells
- the minimum value of `acc[i]` from all active cells
- returns 1 if at least one cell is active

Example 1. The program which computes in each cell:

$$acc[i] \Leftarrow i + \sum_{0}^{p-1} i$$

for $i = 0, 1, \ldots, p-1$ is the following:

```
/**************************************************************************
  index[i] <= index[i] + sum(index[0] + ... + index[p-1]), i = 0, 1, ..., p-1
**************************************************************************/
         cVLOAD(x);       ACTIVATE;  // acc = x = log_2 p; activate all cells
LB(1);   cBRNZDEC(1);     LOAD(5);   // loop (x+2) times; acc[i] <= vectorMem[5]
         cCLOAD(0);       NOP;       // acc <= acc[0] + acc[1] + ... + acc[i] + ...
         cNOP;            CADD;      // acc[i] <= acc[i] + acc
```

The program computes in parallel in $4 + \log 2p$ steps a computation performed by a mono-core in $8p+3$. Therefore, the acceleration is:

$$A = \frac{8p+3}{4+\log_2 p} \in O(p/\log p)$$

In the previous example the acceleration is limited because of the depth of the REDUCTION section of ARRAY. Sometimes, when the reduction operation occurs many times in a sequence of operations, the latency can be avoided and it is added only once at the end of the sequence. This is the case in a very frequently used operation: matrix-vector multiplication.

Example 2. The program for $M \times N$ matrix multiplied by a N-scalar vector, where $M \leq m$ and $N \leq p$ is:

```
/*****************************************************************************
  matrix-vector multiply
*****************************************************************************/
          cNOP;              STORE('MV_W); // mem[i]['MV_W] <= acc[i] = V[i]
          cVLOAD('MV_N);     RLOAD(0);     // acc <= N; acc[i] <= last matrix line
          cVSUB(1);          MULT('MV_W);  // acc <= N-1; acc[i] <= acc[i]* mem[i]['MV_W]
LB(1);    cCPUSHL(0);        RILOAD(255);  // push redSum; acc[i] <= previous line
          cBRNZDEC(1);       MULT('MV_W);  // loop control; acc[i]<=acc[i]* mem[i]['MV_W]
          cVLOAD('MV_S);     NOP;          // load acc for latency loop
LB(2);    cBRNZDEC(2);       NOP;          // latency loop
          cNOP;              SRLOAD;       // load result in acc[i]
```

The execution time is $t_{matrixVectorMultiply} = 2M + 2 + \log_2 p \in 2O(M)$. The mono-core execution is in time belonging to $O(M \times N)$. Then, the acceleration is in $O(N)$.

For $N = p$ the acceleration is in $O(p)$ and, more important, the acceleration is supra-linear because the coefficient of M in $t_{matrixVectorMultiply}$ is 2, and the loop substituted by the parallel execution cannot be performed in 2 cycles only. This performance is obtained because the accelerator performs three processes, executed sequentially in a mono-core engine, in parallel three distinct hardware resources, as follows:

- multiplications in the MAP section of ARRAY
- additions in the REDUCTION section of ARRAY
- the control loop in CONTROLLER

Important notice. The size of ARRAY belongs to $O(p)$. The last hardware evaluation of ACCELERATOR, made for 28 nm technology, provided the following figures for an ARRAY of 2048 32-bit cells, each with 4KB of local memory, running at 1GHz:

- silicon area: 9.2×9.2 mm^2
- power consumption: 12 Watt at 80°C

which translates into: **170 GOPS/Watt**, 24.19 GOPS/mm^2 at 0.14 Watt/mm^2, where *GOPS* stands for **G**iga **O**perations **P**er **S**econd, for 32-bit integer operations.

For flop applications the performance scales down to 170/1.4 GFLOPS/Watt = 121 GFLOPS/Watt. Let us compare our architecture with a Nvidia product implemented in the same 28 nm technology: NVIDIA GK104 GPU[2]. This GPU is a 294 mm^2 chip working at the same frequency and has the following performance: 14 GFLOP/Watt, i.e., $8.64\times$ less performance than our proposal.

How can this big boost in performance generated by our architecture be explained (besides mathematical reasons analyzed in [8] starting from [4])? Let us list few reasons:

- GPU arithmetic is mainly floating-point, while in many applications only integer arithmetic is used.
- For the reduction operations, frequently used in any linear algebra applications, our architecture provides specific hardware which is missing in GPUs.
- Our memory hierarchy is buffer-oriented instead of the cache-oriented architecture used in GPUs. The intense computation accelerated is very predictable regarding to the data and program flow making useless the costly cache mechanism.
- Our cells are execution units, while GPUs cells are processing units which come with the overhead of program memory and its control.
- In our proposal, the control and the execution are performed in parallel in distinct units: accelerator's CONTROLLER and MAP or REDUCTION section.

[2] https://videocardz.net/gpu/nvidia-gk104/

4. EVALUATION

The vectored form of the Floyd-Warshall algorithm, defined on the matrix D, is:

```
/*******************************************
Floyd-Warshall algorithm on vectors
*******************************************/
function FW(D)
 for (k=0; k<N; k=k+1)
  col <= col(k);
  for (j=N; j>0; j=j-1)
   line[j] <= min(line[j],(line[k]+col[0]));
   col <= col << 1;
```

The algorithm is implemented in two versions: (1) for $|V| \leq p$ and for (2) for $|V| = p \times M$.

4.1. The Program for $p \geq |V|$

The maximum input matrix **D** for APSP algorithm is, in this case, a $N \times N$ matrix for $p = N$: The program running on ACCELERATOR is listed in Fig. 2. The execution time, for Floyd-Warshall algorithm with $N \leq p$, on our accelerator is:

$$t_{APSP}(N) = 11.5N^2 + 8N + N\log_2 N + 7 \in O(N^2).$$

```
          cVLOAD(9);      IXLOAD;      // acc <= 9 (= N); acc[i] <= index
          cSTORE(1);      CSUB;        // mem[1] <= N; acc[i] <= index - N
          cVSUB(1);       WHERECARRY;  // acc <= acc - 1; select the first N cells
          cSTORE(4);      NOP;         // mem[4] <= acc (= N-1);
          cVLOAD(0);      NOP;         // acc = 0;
          cSTORE(3);      NOP;         // mem[3] = k
                                       // k LOOP
LB('L1);  cLOAD(3);       IXLOAD;      // acc <= k; acc[i] <= index
          cNOP;           CSUB;        // acc[i] <= acc[i] - acc
          cNOP;           WHEREZERO;   // select the cell k
          cNOP;           VLOAD(255);  // acc[i] <= -1
          cNOP;           ADDRLD;      // addr[i] <= acc[i] (= -1)
          cLOAD(4);       RILOAD(1);   // acc <= mem[4]; acc[i] <= mem[i][addr[i]+1]
                                       // addr[i] <= addr[i] + 1
LB('L2);  cCPUSHL(0);     RILOAD(1);   // {sr[0],sr[1],...}<={acc[k],sr[0],sr[1],...}
          cBRNZDEC('L2);  NOP;         // pc <= acc==0 ? pc+1 : 'L2; acc <= acc - 1
          cLOAD(4);       ENDWHERE;    // acc <= mem[4]; restore previous activation
          cSTORE(2);      NOP;         // mem[2] <= acc (= N-1)
                                       // j LOOP
LB('L3);  cLOAD(3);       NOP;         // acc <= k
          cLOAD(2);       CALOAD;      // acc <= j; acc[i] <= mem[i][k] (= line[k])
          cCSEND(4);      CADD;        // acc[i] <= acc[i] + sr[0]
          cPUSHR(0);      STORE(12);   // {sr[0],sr[1],...}<={sr[1],sr[2],...};
                                       // mem[12] <= acc[i];
          cVSUB(1);       CASUB;       // acc <= acc - 1; acc[i] <= acc[i]-mem[i][k]
          cSTORE(2);      WHERECARRY;  // mem[2] <= acc; select were carry
          cVADD(1);       LOAD(12);    // acc <= acc + 1; acc[i] <= mem[i][12]
          cVSUB(1);       CSTORE;      // acc <= acc - 1; mem[i][j] <= acc[i]
          cBRNZ('L3);     ENDWHERE;    // pc <= acc==0 ? pc+1 : 'L3; restore act.

          cLOAD(3);       NOP;
          cLOAD(2);       CALOAD;
          cCSEND(4);      CADD;
          cNOP;           STORE(12);
          cNOP;           CASUB;
          cNOP;           WHERECARRY;
          cNOP;           LOAD(12);
          cLOAD(3);       CSTORE;
          cVADD(1);       ENDWHERE;
          cSTORE(3);      NOP;
          cSUB(1);        NOP;
          cBRNZ('L1);     NOP;
```

Fig. 2 – The two-column program for APSP.

For comparing “apple with apple” we consider a MRA version implemented in 90 nm, because the literature [2,3] provides data for this technology node. Thus, the table in Fig. 3 compares the performance provided by engines implemented in the same technology. For the acceleration provided by our technology we will compare our implementation in 90 nm with 3.66 GHz Pentium 4 without SSE, at 100 Watt. For each number of vertexes, starting with 64 until 4 096, implementations with $p = 64$ until $p = 4\,096$ are considered (Fig. 3). The values for t_{scalar} represent the execution time associated to the mono-core execution (4.1× than the execution involving the SSE accelerator [2]).

Nodes	t_{scalar}	t_{mra}	α_{mra}^{scalar}	A_{mra}^{scalar}
64	0.00086 s	0.000048 s	17.91	65.57
128	0.00615 s	0.000190 s	32.36	118.46
256	0.04018 s	0.000757 s	53.07	194.26
512	0.21320 s	0.003023 s	70.52	258.12
1024	1.71790 s	0.012077 s	143.08	523.68
2048	14.39100 s	0.048234 s	299.81	1097.31
4096	122.87700 s	0.192137 s	639.98	2342.34

Fig. 3 – Structural and architectural performance for solving APSP problem with Floyd-Warshall algorithm on a mono-core scalar engine (3.66 GHz Pentium 4 without SSE, at 100 W) compared to the run on the MRA architecture where $p \geq N$.

We conclude that the use of the peak performance of our MRA accelerator remains > 50%. More important: we compare an 100 Watt engine with our engine which for $p = 1\,024$, for example, is powered with less than 30 Watt and provides > 512× acceleration.

4.2. The performance for $p = 128 < |V|$

Let us consider now $|V| = M \times N$ and $p = N$. An MRA accelerator with $p = 128$ cells (90 nm, at 1GHz working at 5 Watt) is compared with 1.35 GHz, 128-core NVidia Quadro FX5600 working at 178 Watt (Fig. 4). The running time for this case is:

$$t_{APSP}(M,N) \simeq M^2N^2(11.5M - 0.25) \in O(N^2M^3)$$

and we obtain the expected acceleration in $O(p)$.

Nodes	t_{gpu}	t_{mra}	α_{mra}^{gpu}	A_{mra}^{gpu}
64	0.0014 s	0.000048 s	29.16	39.36
128	0.0025 s	0.000190 s	13.15	17.75
256	0.0077 s	0.001490 s	5.16	6.92
512	0.0349 s	0.011993 s	2.91	3.92
1024	0.2301 s	0.096206 s	2.39	3.22
2048	1.7356 s	0.770703 s	2.24	3.02
4096	13.7200 s	6.169821 s	2.22	2.99

Fig. 4 – Structural and architectural performance for solving APSP problem with Floyd-Warshall algorithm on a GPU (1.35 GHz, 128-core NVidia Quadro FX5600 at 178W) compared to the run on the MRA architecture where $P = N = 128$ (128-core at 5W).

In Fig. 4 the APSP problem is solved with two engines, each of 128 cells. Results are in favor for our proposal, which has at least 3× architectural performance at 26× less energy.

5. CONCLUSIONS

1. The acceleration provided by a p-cell implementation of the proposed architecture is in $O(p)$ for APSP problem solved using the Floyd-Warshall algorithm. The performance of our accelerator is >3× higher than the performance provided by GPUs used as general-purpose accelerators.

2. Our proposal offers also a spectacular energy saving: more than 20× less energy for the same computation.

3. The main reason for the improvements offered by the MRA accelerator is due to the architectural freedom allowed by the lack of any embarrassing software legacy. Instead of using *ad doc* structured parallel machines (see Intel's MIC) or application-oriented structures (like GPUs), both marked by heavy legacies, we are able to start fresh and with a solid mathematical foundation, free of any corporate pressure.

4. The only drawback is related with the programming environment which is now limited to the use of an accelerated library based on a kernel written in assembly language and developed at the library level using a standard general-purpose language (for example: the *Eigen* kernel implemented on the accelerator's kernel is used to write in C++ the *Eigen Library*).

REFERENCES

1. V. DRAGOMIR, *All-pair shortest path modified matrix multiplication based algorithm for a one-chip MapReduce architecture*, U.P.B. Sci. Bull., Series C, **78**, *4*, pp. 95-108, 2016
2. S.C. HAN, F. FRANCHETTI, M. PUSCHEL, *Program generation for the all-pairs shortest path problem*, Parallel Architectures and Compilation Techniques (PACT), pp. 222-232, 2006, http://users.ece.cmu.edu/~franzf/papers/pact06.pdf
3. G.J. KATZ, J.T. KIDER. *All-pairs shortest-paths for large graphs on the GPU*, Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS, Symposium on Graphics Hardware (GH'08), pp. 47-55, 2008. Available online: https://repository.upenn.edu/cgi/viewcontent.cgi?article=1213&context=hms
4. S. KLEENE, *General recursive functions of natural numbers*, Mathematische Annalen, **112**, *1*, pp. 727-742, 1936.
5. V. KUMAR, A. GRAMA, A. GUPTA, G. KARYPIS, *Introduction to parallel computing: design and analysis of algorithms*, The Benjamin/Cummings Publishing Company, 1994.
6. M. MALITA, G.M. STEFAN, D. THIEBAUT, *Not multi-, but many-core: Designing integral parallel architectures for embedded computations*, ACM SIGARCH Computer Architecture News, **35**, *5*, pp. 32-38, 2007.
7. G.M. STEFAN, A. SHEEL, B. MITU, T. THOMSON, D. TOMESCU, *The CA1024: A fully programmable system-on-chip for cost-effective HDTV media processing*, Stanford University: Hot Chips – A Symposium on High Performance Chips, August 2006. Available online: https://youtu.be/HMLT4EpKBAw at 35:00.
8. G.M. STEFAN, M. MALITA, *Can one-chip parallel computing be liberated from ad hoc solutions? A computation model based approach and its implementation*, 18th Inter. Conf. on Ciruits, Systems, Communications and Computers, Santorini (Greece), July 2015, pp. 582-597, http://users.dcae.pub.ro/~gstefan/2ndLevel/technicalTexts/COMPUTERS2-42.pdf

Received August 7, 2019