BLAST on UPMEM

Dominique Lavenier, Charles Deltel, David Furodet, Jean-François Roy

BLAST on UPMEM

Dominique Lavenier, Charles Deltel,
David Furodet, Jean-François Roy

RESEARCH
REPORT

N° 8878

March 2016

Project-Team GenScale

ISSN 0249-6399

BLAST on UPMEM

Dominique Lavenier[1], Charles Deltel[1]
David Furodet[2] , Jean François Roy[2]

Project-Team GenScale

Research Report N° 8878 — March 2016 —19 pages.

Abstract: This paper presents the implementation of the BLAST software on the UPMEM architecture. BLAST is a well-known molecular biology software to rapidly scan DNA or protein genomic banks. It is daily used by thousands of biologists. Given a query sequence, BLAST reports all the sequences of the bank where similarities are found. The result is a list of *alignments* that detail the similarities. UPMEM's Processing-In-Memory (PIM) solution consist of adding processing units into the DRAM, to minimize data access time and maximize bandwidth, in order to drastically accelerate data-consuming algorithms. A 16 GBytes UPMEM-DIMM module comes then with 256 UPMEM DRAM Processing Units (named DPU). To find similarities, BLAST proceeds in 3 steps: (1) search of common words between the query sequence and the bank sequences; (2) evaluation of local similarity on the neighbourhood of these words; and (3) computation of the final alignment. As the 2 first steps are limited by memory bandwidth, and represent the majority of time of the overall computation, they have been massively parallelized on UPMEM DPUs. The 3rd step is performed on the host processor and is overlapped with the UPMEM processing.

Experimentation on real datasets shows a speed-up of 25 when using UPMEM configuration, versus a standard server running 20 Intel cores.

Key-words: BLAST, genomic banks, Processing-in-Memory

[1] INRIA / IRISA – dominique.lavenier@inria.fr
[2] UPMEM SAS – jroy@upmem.com

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

BLAST on UPMEM

Résumé : Ce rapport présente l'implémentation du logiciel BLAST sur l'architecture UPMEM. BLAST est un logiciel bien connu en biologie moléculaire pour scanner les banques de séquences protéiques ou d'ADN. Il est utilisé par de milliers de personnes chaque jour. Etant donné une séquence requête, BLAST retourne toutes les séquences de la banque où des similarités ont été détectées. Le résultat est une liste *d'alignements* qui détaillent ces similarités. La solution de *Processing-in-Memory* (PIM) UPMEM consiste à ajouter des unités de calcul dans la DRAM, pour minimiser le temps d'accès aux données et maximiser la bande passante, de manière à accélérer significativement les algorithmes gourmands en données. Une barrette UPMEM de 16G Mo propose donc 256 DRAM *Processing Units* (appelé DPU). BLAST décompose le calcul de recherche de similarité en 3 étapes : (1) recherche de mots communs entre la séquence requête et les séquences de la banque, (2) recherche locale de similarité autour de ces mots, et (3) calcul de l'alignement final. Comme les 2 premières étapes sont limitées par la bande-passante du serveur, et représentent la majorité du temps de calcul, elles ont été implémentées de manière massivement parallèle sur des DPUs UPMEM. La 3ème étape, quant à elle, est réalisée sur le processeur hôte en recouvrement des calculs sur UPMEM.

Des expérimentations sur des jeux de données réels, montrent un facteur d'accélération de 25 avec la configuration UPMEM, par rapport à un serveur traditionnel équipé de 20 cœurs Intel.

Mots clés : BLAST, banques génomique, Processing-in-Memory

1. Blast Software

BLAST is a popular bioinformatics software aiming at finding similarities between DNA or protein sequences [1][2]. BLAST is used every day by thousands of biologists or bioinformaticians to scan the genomic databases or to perform intensive sequence comparison between large genomic data sets. It is a fundamental processing tool, used in many bioinformatics pipelines. BLAST software consists of a family of software, each one dealing with different types of sequences.

With the fast evolution of sequencing techniques, BLAST (as other software) must face with the explosion of data. Comparing large sets of data coming from sequencing projects with regular standard banks is becoming extremely time-consuming. And, in more and more cases, becomes a serious bottleneck. There is a strong need to speed up this task.

The primary goal of BLAST is to search genomic banks to report similarities between sequences. More precisely, from a set of *query* sequences, BLAST reports all *alignments* between theses sequences and all the sequences of a *reference* bank. The output of BLAST is a list of local alignments, as represented below:

```
>sp|P33501|CYB_ANOQU Cytochrome b OS=Anopheles quadrimaculatus

 Score =   122 bits (307),  Expect = 1e-28
 Identities = 77/206 (37%), Positives = 106/206 (51%), Gaps = 21/206 (10%)

Query  1094  YIFYFLLLLQVKFGVHLLFLHETGSNNPLGVRRDLDKLPFHPYFSVKDLFGGFCYTWATF  1153
             +IF F++L  +   +HLLFLH+TGSNNPLG+  ++DK+PFHPYF  KD+FG   + W
Sbjct  184   FIFPFIILALMM--IHLLFLHQTGSNNPLGLNSNVDKIPFHPYFIYKDIFGFIVFYWILI  241

Query  1154  -----FESALLPPELXETQKILFLLIPXLP--LFIFNQSGIFXWPMLFYG--QFLISXEV  1204
                   F   L+ PE       L   +   P   F+F  + +   P    G    ++S  +
Sbjct  242   RFIWKFNYLLMDPENFIPANPLVTPVHIQPEWYFLFAYAILRSIPNKLGGVIALVLSIAI  301

Query  1205  XLHLPFLFLSXXYVLLXKKVIFGDXPFYPLSKFYFXIHVNTXILLTXIGARPVEDPYITI  1264
              L LPF   S           F    FYPL++  F   V    LLT IGARPVEDPY+
Sbjct  302   LLILPFTHSSK----------FRGLQFYPLNQILFWNMVVVASLLTWIGARPVEDPYVLT  351

Query  1265  GQIFSAVYFLFYFLNPFLIGLXDKXI  1290
             GQI + +YF ++ +NP L    DK +
Sbjct  352   GQILTVLYFSYFIINPLLAKYWDKLL  377
```

A local alignment details the way two portions of DNA or protein sequences are matching. Attached to an alignment is an expected value (probabilistic information) allowing BLAST to select only relevant ones. When large genomic banks are processed, an alternative output format summarizing the features of the alignment is preferred. This compressed format is better suited for post processing tasks.

The BLAST algorithm is based on a powerful heuristic, assuming that relevant alignments contain similar sequences with common words on N characters. These words are called *seeds* and are primarily searched. The structure of the BLAST algorithm can thus be decomposed into 3 main steps:

1. Seed search
2. Seed extension
3. Alignment computation

To search *seeds*, the query sequence is first indexed on N-word characters basis. Then, the reference bank is scanned, words by words, to find similar N-words. When there is a match between a query and a

reference word, the neighborhood of the seed is explored, as step 2. This step simply computes a local similarity. If it exceeds a threshold value, then a complete alignment is computed.

Profiling the BLAST code indicates that most of the computation time is performed in step 1 & 2. As a matter of fact, in many cases, the match of two seeds arises by chance and does not lead to any alignments. Step 2 works as a filter that rejects most of the match proposals.

Comparing one query sequence with a full reference bank generates millions of seed matches. **But all these matches can be processed independently**. In other words, there is a great opportunity for massive parallelism, as offered by the UPMEM technology.

2. UPMEM Architecture Overview

UPMEM technology is based on the concept of Processing-in-Memory (PIM). The basic idea is to add processing elements next to the data, i. e. in the DRAM, to maximize the bandwidth and minimize the latency. Host processor is acting as an orchestrator: It performs read/write operations to the memories and controls the co-processors embedded in the DRAM. This data-centric model of distributed processing is optimal for data-consuming algorithms.

A 16 GBytes UPMEM DIMM module comes with 256 processors: One processor every 64 MBytes of DRAM. Each processor can run its own independent program. In addition, to hide memory latency, these processors are highly multithreaded (up to 24 threads can be run simultaneously) in such a way that the context is switched at every clock cycle between threads.

The UPMEM DPU is a triadic RISC processor with 24 32-bits registers per thread. In addition to memory instructions, it comes with built-in atomic instructions and conditional branching bundled with arithmetic and logic operations.

From a programing point of view, two different programs must be specified: (1) the host program that will dispatch the data to co-processors memory, sends commands, input data, and retrieve the results; (2) the program that will specify the treatment on the data stored in the DPU memory. This is often a short program performing basic operations on the data. It is called a *tasklet*. Note however, that the architecture of the UPMEM DPU allows different tasklets to be specified and run concurrently on different blocks of data.

Depending on the server configuration (i. e. the number of 16 GBytes UPMEM modules), a large number of DPU can process data in parallel. Each DPU only accesses 64 MBytes and cannot directly communicate with its neighbors. Data exchanges, if needed, must go through the host processor. A DPU has a fast working memory (64 Kbytes) acting as cache/scratchpad memory and shared by all tasklets (threads) running on the same DPU. This memory working space can be used to transfer blocks of data from the DRAM, and can be explicitly managed by the programmer.

To sum up, programing an application consists in writing a main program (run on the host processor) and one or several tasklets that will be executed on the DPUs. The main program has to synchronize the data transfer to/from the DPUs, as well as the tasklet execution. Note that the tasklet execution can be run asynchronously with the host program, allowing host tasks to be overlapped with DPU tasks.

3. BLAST implementation on UPMEM

In this section, we present the implementation of BLASTp/BLASTx that compares a set of query proteins or DNA sequences with a protein databank. The difference between BLASTp and BLASTx is that BLASTx first translates the DNA sequences into the six possible protein reading frames.

As introduced in section 1, the BLAST algorithm can be decomposed into 3 steps: (1) seed search; (2) seed extension; (3) alignment computation. Only the two first steps, that are the most time consuming, are offloaded to the UPMEM DPUs.

The protein bank is split across all DPUs. Hence, each DPU stores only a part of the protein bank. Every query is broadcasted to all DPUs to perform both the search seed step and the search extension step. Inside a DPU, the task is again split into similar subtasks that will only have to look after a restricted part of the protein bank devoted to that DPU. The general implementation can be described as follows:

```
Dispatch the reference bank inside DPU
For each query sequence:
   Index the query
   Broadcast the query and its index to the DPUs
   Perform parallel scanning on UPMEM
   Get results from UPMEM
   Compute alignments
```

Dispatching the reference bank into the DPUs is done in a round-robin fashion to avoid burst of similar sequences into the same DPU. This is done once, at the beginning of the treatment. Data are read from the disk and the reference bank is stored both on the host DRAM and on the UPMEM DRAM. Actually, in the current implementation, the host processor needs to completely access the bank to perform the final step. Storing the bank only on the UPMEM side would require extra time dedicated to data exchange, an overhead estimated to 10% of the overall computation. The advantage of the current implementation is that the time devoted to exchange the data between the host processor and all the DPUs is negligible compared to the effective computation time (see host/DPU exchange estimation time in section 4).

For each query, an index is computed. Figure 1 details the index structure. A first table (Index1) memorizes for each seed its number of occurrences in the query (N part) and an entry number (P part) in a second table (index2). This second table stores the coordinates where the seed occurs in the query.

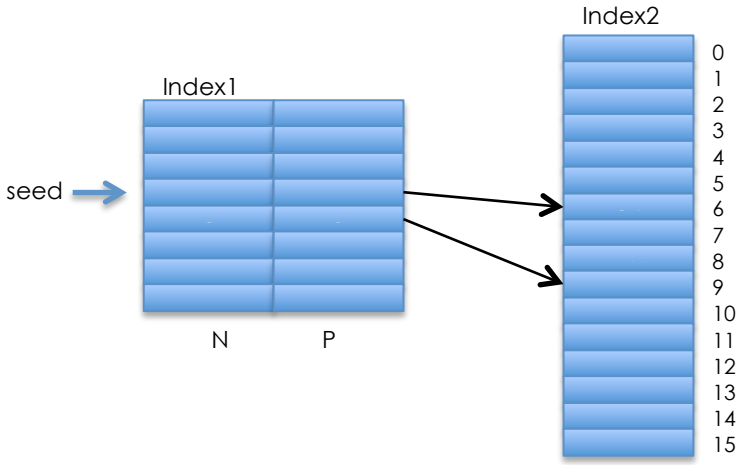


Figure 1: structure of the query index

For performance purpose, the index must fit in the UPMEM working memory and has a limited size. The query sequences must not exceed 3000 characters. Fortunately, the average size of a protein sequence range from 300 to 400 amino acids. We then optimized the process by packing several sequences in the index when possible.

The query sequence (or the group of queries) and its index are then broadcasted to all DPUs. Each DPU starts its tasklets that perform independent searches. As in BLAST, the reference is scanned word by word. Each word queries the index that indicates (or not) a potential match (step 1 of the algorithm). If so, the second step is fired and a score corresponding to a sequence similarity is calculated (see tasklet program below). When the score exceeds a predefined threshold value, coordinates of both the query and reference sequences are memorized in a list. At the end of the scan, each tasklets send back this list that represents the coordinates of all successful seed-extensions.

When all DPUs have finished, the host computes the final alignments based on this list of coordinates. Note that this execution scheme is sequential: step 1 & 2 need to be performed before step 3 starts. To maximize the overall throughput an overlap strategy is implemented: As there are many queries to process, overlapping step 1 & 2 with step 3 is straightforward. The total execution time is thus given by the maximum of step (1 + 2) and step 3.

Tasklet program

The tasklet program performs step 1 & 2. It scans, sequence by sequence, a small part of the protein bank. A sequence is systematically decomposed into words of N characters (seeds) that are compared to the query index. For each seed match, a local similarity is computed, by counting the number of identical characters in a restricted neighborhood (here, a window of 16 characters). If the number of identical characters exceeds a threshold value, then the coordinates of both the query and the bank sequences are sent as results. Below is a simplified version of the algorithm. The complete code is given in annex 1.

```
Input :  seqBank[]   // array of sequences
         seqQuery    // query sequence

Output : RESULTS[]   // array to store coordinates

r = 0 ;
for (nr = 0 ; nr < len(seqBank) ; nr++)
 {
   for (ir = 0 ; ir < len(seqBank[nr])-sizeNeigborhood ; ir++)
    {
      seed = makeSeed(seqBank[nr],ir)
      n = INDEX1_N[seed]
      p = INDEX1_P[seed]
      for (k=0 ; k<n ;k++)
       {
         iq = INDEX2[p+k]
         sc = similarity_score(seqQuery,iq,seqRef,ir) ;
         if (sc > Threshold)
           {
             RESULT[r++] = (iq,ir,nr) ;
           }
       }
  }
```

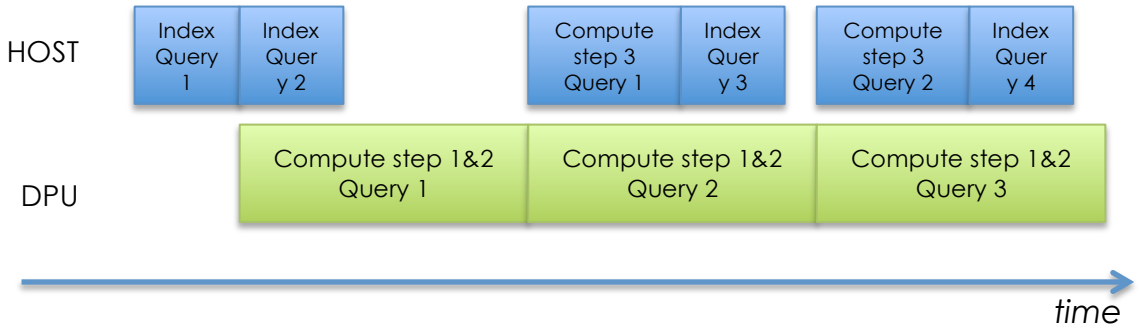
Alignment program

The alignment program performs step 3 and is executed on the host processor. It receives from all the DPUs a list of coordinates where significant seed extensions have been found between the query sequence and some sequences of the bank. From these coordinates, a banded Smith & Waterman algorithm [3][4] is executed. Only significant alignments with a low e-value (user parameter) are

reported. To speed-up the process, this task is optimized with SSE instructions and run in parallel using all available cores of the processor.

HOST/DPU task overlapping

When a large set of query sequences are compared to a reference bank, host and DPU tasks can be advantageously overlapped as shown below. The host has the charge of computing the query index and the final alignment step, while the DPUs perform a parallel scanning of the reference bank. In that case, the overall execution time will be constrained by the longest task. An optimal implementation will have to provide a good balance between these tasks that can be adjusted either by the number of DPUs or the number of processor cores.



4. Performance evaluation

Speed up evaluation

Performances of the BLAST implementation have been evaluated on a DELL server (Xeon Processor E5-2670, 40 cores 2.5 GHz, 64 GBytes RAM) running Linux Fedora 20.

The two following dataset have been considered:

	Query bank	Reference bank
Dataset 1	Arabidopsis Thaliana proteom 31477 proteins	SwissProt, version xxx 545 536 proteins (194 x10^6 AA)
Dataset 2	Marine meta-transcriptomic 1000 RNA sequences	RefSeq, version 74 50 496 795 proteins (18 x 10^9 AA)

For dataset 1, the ncbi BLASTp program (version 2.2.29+) has been run with 20 threads and with an e-value set to 10^{-6}:

```
blastp -query query_bank -db ref_bank -evalue 1e-6 -comp_based_stats F
       -outfmt 6 -max_target_seqs 500 -num_threads 20 -out query_ref.m8
```

For dataset 2, the ncbi BLASTx program (version 2.2.29+) has been run with 20 threads and with an e-value set to 10^{-6}:

```
blastx -query query_bank -db ref_bank -evalue 1e-6 -comp_based_stats F
       -outfmt 6 -max_target_seqs 500 -num_threads 20 -out query_ref.m8
```

The table below gives the elapsed execution time reported with the Linux time command.

	Dataset 1 - BLASTp	Dataset 2 - BLASTx
BLAST Exec. time	2h05	1h40

The execution time of the tasklets (step 1 & 2) is estimated from the Cycle Accurate Simulator (CAS) developed by the UMEM Company. The execution time of the banded Smith & Waterman algorithm (step 3) is directly measured from an optimized code developed by the GenScale IRISA/INRIA research group.

Dataset 1

# DPU	512	1024	2048	4096
DPU exec. time (sec)	660	330	165	82
HOST exec. Time (sec)	300	300	300	300
Speed up	11.3	**22.7**	**25**	25

The DPU frequency is set to 750 MHz
The speed-up is given by: Blast time / max (DPU time, HOST time)

The speed up cannot exceed 25 because the host contribution is the limiting factor for this dataset.

On this dataset, in average, a DPU requires 280 cycles to process one amino acid (AA) on the reference bank, that is 373 ns. Knowing that the reference bank is equitably split over all the DPUs, the overall DPU execution time is given by:

$$\text{DPU execution time} = \text{NB\_QUERY} \times \text{NB\_AA\_REF} / \text{NB\_DPU} \times 373 \text{ ns}$$

NB_QUERY is the number of blocks of queries sent to each DPU (4658)
NB_AA_REF is the number of amino acids of the reference bank (194×10^6)

We also need to add the communication time between the host and the DPUs. Intel mlc tool has been used to evaluate the memory bandwidth of the server (see detail on Annex 2). We choose the 3:1 Reads/Writes configuration providing a bandwidth of 46.8 GBytes/sec. Host/DPU communication can be decomposed into 4 steps as described below (annex 3 details each steps).

1 - Load of the database (only once at the beginning of the process). The reference bank load time (BT) is given by:

$$BT = \text{SIZE\_REF\_BANK} / \text{MEMORY\_BANDWIDTH} = 194 \times 10^6 / 46.8 \times 10^9 = \mathbf{4.14\ ms}$$

- SIZE_REF_BANK is the size of the reference bank. One amino acid is stored on 1 Byte.

2 – Load of the queries. Several queries are packed together in order to optimize the query index. The size of the index is equal to 6.4 Kbytes. The same index is broadcasted to all DPUs. The query load time (QT) is given by :

$$QT = (\text{NB\_BLOCK} \times \text{SIZE\_INDEX} \times \text{NB\_DPU})/\text{MEMORY\_BANDWIDTH}$$
$$QT = (4658 \times 6400 \times 1024)/46.8 \times 10^9 = \mathbf{652\ ms}$$

3 – Load of parameters needed by tasklets. Two integers are sent for each tasklet. The parameter load time (PT) is given by:

$$PT = (\text{NB\_BLOCK} \times \text{NB\_DPU} \times \text{NB\_TASKLET} \times 8)/\text{MEMORY\_BANDWIDTH}$$
$$PT = (4658 \times 1024 \times 10 \times 8)/46.8 \times 10^9 = \mathbf{8.15\ ms}$$

4 – load of the results. Each DPU send back to the host the coordinates (2 integers) of the sequences when a match has been detected. In average, for this dataset, and for a block of queries a DPU generate 200 couples of integers. The results transfer time (RT) is given by:

$$RT = (NB\_BLOCK \times NB\_DPU \times 200 \times 8) / 46.8 \times 10^9 = \mathbf{163\ ms}$$

The complete host/DPU communication time is equal to: BT + QT + PT + RT = **828 ms**. It is included in the host execution time in the above table.

Dataset 2

# DPU	512	1024	2048	4096
DPU t execution time (sec)	1680	840	420	210
HOST execution time (sec)	180	180	180	180
Speed up	3.5	7.1	14.2	**28.5**

The DPU frequency is set to 750 MHz
The speed-up is given by: Blast time / max (DPU time, Host time)

The speed up maximum is 28.5. The number of DPU is the limiting factor for this dataset.

On this dataset, in average, a DPU requires 220 cycles to process one amino acid (AA) on the reference bank, that is 293 ns. Knowing that the reference bank is equitably split over all the DPU, the overall DPU execution time is given by:

$$DPU\ execution\ time = NB\_QUERY \times NB\_AA\_REF / NB\_DPU \times 293\ ns$$

NB_QUERY is the number of blocks of queries sent to each DPU (163)
NB_AA_REF is the number of amino acids of the reference bank (18×10^9)

The Host/DPU computation time is evaluated using the same bases as Dataset 1 and provide the following times:

BT = SIZE_REF_BANK / MEMORY_BANDWIDTH = $18 \times 10^9 / 46.8 \times 10^9$ = **385 ms**
QT = NB_BLOCK x SIZE_INDEX x NB_DPU)/MEMORY_BANDWIDTH = **24.2 ms**
PT = (NB_BLOCK x NB_DPU x NB_TASKLET x 8)/MEMORY_BANDWIDTH = **0.3 ms**
RT = (NB_BLOCK x NB_DPU x 300 x 8) = **9 ms**

Quality Evaluation

Alignments computed by the UPMEM version of BLAST have been compared with the BLAST NCBI software. Results slightly differ for the following reasons:

- The seed search step considers words of only 2 amino acids only. The seed size of BLAST is 3, but is leveraged by the substitution matrix. The UPMEM implementation is thus more sensitive at this step.
- The seed extension step is restricted to a window of 16 amino acids right next to the seed. It counts the number of identical pseudo amino acids. The amino acid alphabet has been reduced to 14 according to the work of [5]. When a specific number of matches is detected, we consider that a complete alignment computation can be fired.
- The alignment computation is done on restrictive banded Smith & Waterman algorithm using the

SSE technology available on today processors. The size of the band depends of the technology: SSE4, AVX and AVX2 provide respectively a band size of 16, 32 or 64. BLAST has no such restriction. For fragmented alignments (i.e. alignment with many long gaps) the impact is that several alignments are generated instead of a single one.

Overall, the sensitivity of UPMEM BLAST is comparable to the NCBI BLAST. Both of them are based on the same general heuristic. However, step 1 & 2 of UPMEM BLAST have better sensitivity, but it can be slightly damaged by the SSE constraint of step 3. Practically, both software find 95 to 97 % of identical alignments. The rest belong to the grey zone depending of many optimization threshold acting all along the production of an alignment, including statistical consideration allowing an alignment to be discarded or not.

5. Comparison with CUDA BLASTP

CUDA BLASTP is one of the best implementation of BLAST on GPU. In [6], the authors test their implementation with the NCBI BLAST software by comparing protein sequences of different length with the GenBank Non-redundant Protein Database (3,163,461,953 Amino Acids in 9,230,955 Sequences).

The GPU board has the following features: NVIDIA GeForce GTX 295: dual GPU, 1,242 MHz processor clock speed, 60 SMs (30 per GPU), 1,792 MB GDDR3 device memory (896 MB per GPU). Tests have been conducted with these cards installed in a PC with an Intel Quad-Core i7-920 2.66 GHz CPU, 12 GB RAM running Linux Fedora 10.

The following table reports the performance speed up of GPU for different query sizes:

Query size (AA)	127	254	517	1054	2026
NCBI BLAST Exec. Time (sec)	34.5	36.1	63.8	131.7	155.3
CUDA-BLASTP Exec. Time (sec)	5.8	5.9	11.9	37.7	29.2
Speed up	5.9	6,1	5.3	3.5	5,3

Estimation on a 512-DPU UPMEM system

We can estimate the performances of a 512 DPU architecture as follows:

DPU execution time = NB_AA_REF / NB_DPU x T_AA

The number of cycles for a DPU to process one amino acid of the bank depends of the query size. There is a fixe cost to scan the reference (about 150 cycles) then a cost that linearly depends of the query size, up to 250 cycles for the largest sequence (2026 AA). Thus T_AA = NB_CYCLE / 0.75 x 10^9 (frequency set to 750 MHz).

For the host execution time, we apply again a ratio of 1/25 on the NCBI BLASTP execution time.

The following table gives an estimation of the UPMEM execution time (and speed up compared to NCBI BLAST) on the dataset used by the authors of CUDA-BLASTP.

Query size (AA)	127	254	517	1054	2026
NCBI BLAST Exec. Time (sec)	34.5	36.1	63.8	131.7	155.3
Time to scan one AA (ns)	208	216	234	270	333
DPU (sec)	3.73	3.88	4.2	4.85	6
Host (sec)	1.4	1.5	2.55	5.26	6.2
Speed up	**9.24**	**9,30**	**15,2**	**25**	**25**

Bibliography

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W. & Lipman, D.J. (1990) "Basic local alignment search tool." J. Mol. Biol. 215:403-410.

2. Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W. & Lipman, D.J. (1997) "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res. 25:3389-3402.

3. Smith, Temple F.; and Waterman, Michael S. (1981). "Identification of Common Molecular Subsequences" (PDF). Journal of Molecular Biology **147**: 195–197.

4. Osamu Gotoh (1982). "An improved algorithm for matching biological sequences". Journal of molecular biology **162**: 705

5. Tanping Li, Ke Fan, Jun Wang, and Wei Wang, Reduction of protein sequence complexity by residue grouping Protein Eng. (2003) 16 (5): 323-330

6. Weiguo Liu, Bertil Schmidt, and Wolfgang Muller-Wittig. 2011. CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 8, 6 (November 2011), 1678-1684. DOI=http://dx.doi.org/10.1109/TCBB.2011.33

Annex 1: Tasklet Program

```
//
// TASKLET PROGRAM FOR SEED-EXTENSION ALGORITHM
//
// The bank is spilt over different DPUs in the MRAM
// The bank is stored as a list of protein sequences
// Each tasklet has a portion of the bank to process
//


#include <stdint.h>
#include <alloc.h>
#include <mram_ll.h>
#include "se.h"
#include "mbox.h"
#include "shm.h"
#include <devprivate.h>
#include <meetpoint.h>
#define USE_BUILTINS
#ifdef USE_BUILTINS
#include <built_ins.h>
#endif

int searchDB(uint32_t offset_bank, int *idxp, int *idx, int  *dq,
             int8_t *buffer_bank, int *buffer_res, int offset_res)
{
  int i, ib, k, kk, k1, k2, key, n, p, j, iq, ir, score, num_seq, nb_score, addr_res;
  unsigned int R0, R1, R2, R3, T, S;

  addr_res = offset_res;
  num_seq = -1;
  nb_score = 0;

  mram_ll_read256(offset_bank,buffer_bank);
  offset_bank = offset_bank+256;

  R0 = 0;
  for (i=5; i>=2; i--) R0 = (R0<<8)+(buffer_bank[i]&0xF);
  R1 = 0;
  for (i=9; i>=6; i--) R1 = (R1<<8)+(buffer_bank[i]&0xF);
  R2 = 0;
  for (i=13; i>=10; i--) R2 = (R2<<8)+(buffer_bank[i]&0xF);
  R3 = 0;
  for (i=17; i>=14; i--) R3 = (R3<<8)+(buffer_bank[i]&0xF);


  ib=0; ir=0;
  while (1)
    {
      if ((buffer_bank[ib]&0xF0) > 0)
        {
          if (buffer_bank[ib]&0x10) { num_seq=num_seq+1; ir=0; }
          else if (buffer_bank[ib]&0x20) break;
        }
      key = (buffer_bank[ib]&0xF) + ((buffer_bank[ib+1]&0xF)<<4);
      n = idxp[key] >> 16;
      p = idxp[key] & 0xFFFF;
      for (k=n; k>=0; k--)
       {
         iq = idx[p+k];

         __builtin_cmpb4_rrr(T, R0, dq[iq]);
         if (T==0) continue;
         __builtin_cao_rr(score, T);  // score <= number of ones in T
         __builtin_cmpb4_rrr(T, R1, dq[iq+4]);
         __builtin_cao_rr(S, T);
         score += S;
         if (score < 3) continue;
         __builtin_cmpb4_rrr(T, R2, dq[iq+8]);
         __builtin_cao_rr(S, T);
         score += S;
```

```
          __builtin_cmpb4_rrr(T, R3, dq[iq+12]);
          __builtin_cao_rr(S, T);
          score += S;

          if (score >= 8)
            {
              buffer_res[nb_score&0xF] = (score<<16) + num_seq;
              nb_score++;
              buffer_res[nb_score&0xF] = (ir<<16) + (iq-2);
              nb_score++;
              if ((nb_score&0xF) == 0)
               {
                 mram_ll_write64(buffer_res,addr_res);
                 addr_res = addr_res+64;
               }
            }
        }
      ir++; ib++;
      if (ib>=256-BANK_OVERLAP)
       {
         mram_ll_read256(offset_bank,buffer_bank);
         offset_bank = offset_bank+256;
         ib=0;
       }
      R0 = (R0>>8)|((R1&0xF)<<24);
      R1 = (R1>>8)|((R2&0xF)<<24);
      R2 = (R2>>8)|((R3&0xF)<<24);
      R3 = (R3>>8)|((buffer_bank[ib+17]&0xF)<<24);
    }
  mram_ll_write64(buffer_res,addr_res);
  return nb_score;
}
```

Annex 2: Intel MLC report

```
[root@koriscale (Fedora 20) Linux]$ ./mlc
Intel(R) Memory Latency Checker - v3.0
Measuring idle latencies (in ns)...
                Numa node
Numa node            0       1
       0         64.6   117.7
       1        118.2    62.8

Measuring Peak Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      47698.6
3:1 Reads-Writes :      46813.3
2:1 Reads-Writes :      57056.4
1:1 Reads-Writes :      57377.0
Stream-triad like:      38628.1

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        46934.7 23535.5
       1        23288.1 46691.6

Measuring Loaded Latencies for the system
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
Inject  Latency Bandwidth
Delay   (ns)    MB/sec
==========================
 00000  519.52    42193.1
 00002  519.83    42359.0
 00008  522.82    42439.1
 00015  524.55    42564.1
 00050  531.52    42749.9
 00100  532.44    42623.7
 00200  506.70    42977.4
 00300  230.56    38555.2
 00400  180.66    30051.4
 00500  166.83    24531.3
 00700  152.37    18010.2
 01000  146.45    12917.4
 01300  142.61    10120.3
 01700  140.14     7891.2
 02500  138.62     5549.2
 03500  137.93     4115.9
 05000  134.01     3041.6
 09000  130.32     1919.5
 20000  129.13     1140.6

Measuring cache-to-cache transfer latency (in ns)...
Local Socket L2->L2 HIT  latency        28.0
Local Socket L2->L2 HITM latency        31.6
Remote Socket LLC->LLC HITM latency (data address homed in writer socket)
                Reader Numa Node
Writer Numa Node     0       1
          0       -    73.1
          1     73.0       -
Remote Socket LLC->LLC HITM latency (data address homed in reader socket)
                Reader Numa Node
Writer Numa Node     0       1
          0       -    72.6
          1     72.9       -
```

Annex 3: Host/DPU data transfer

A complete database search requires the following data transfers:

1. Load the data base into DPU memory (once for the whole process)
2. Load of the queries
3. Send parameters to DPU
4. Get the results from each DPUs

Steps 2, 3 and 4 are repeated N times. N is the number of packets of queries (several queries are grouped to minimized data transfer).

```
// load the bank (time = BT)
for (nd=0; nd < NB_DPU; nd++)
   copy_to_dpu(dpu[nd], BANK, ADDR_BANK, SIZE_BUFFER*sizeof(int8_t));


// load group of queries and the associated indexes (time = QT)
for (nd=0; nd < NB_DPU; nd++) {
    copy_to_dpu(nd,  256 x sizeof(int));
    copy_to_dpu(nd, 3072 x sizeof(int));
    copy_to_dpu(nd, 3072 x sizeof(int));
  }

// send information to DPU through the mailbox (time = PT)
for (nd=0; nd < NB_DPU; nd++)
   for (nt=0; nt < NB_TASKLET; nt++)
      dpu_post(nd,nt,2*sizeof(int));


// get results (time = RT)
for (nd = 0; nd < NB_DPU; nd++)
   for (nt=0; nt<NB_TASKLET; nt++)
      {
        dpu_receive(nd,nt2*sizeof(int));
        copy_from_dpu(nd,nt,N*sizeof(int));
      }
```

The following table summarizes the different transfer times for the two datasets when considering a bandwidth of 46.8 GBytes/sec

	BT (ms)	QT (ms)	PT (ms)	RT (ms)	Total (ms)
Dataset 1	4.14	652.00	8.15	163.00	828.00
Dataset 2	385.00	24.20	0.30	8.00	417.50

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399