

# PGAS: Partitioned Global Address Space

presenter: Qingpeng Niu

January 26, 2012

# Outline

# Motivation

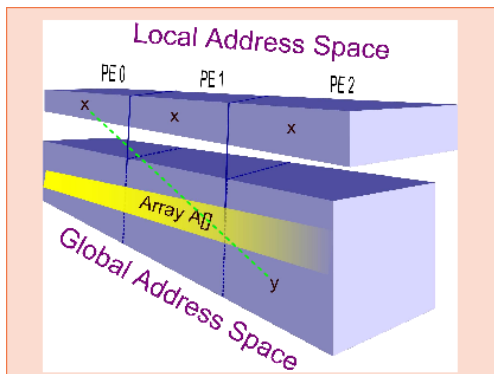
1. Large scale cluster is everywhere.
2. Shared memory programming model with easy programming and more productivity.
3. MPI with hard programming but high performance and scalability
4. Sometimes memory can not be hold by node for large applications.

## PGAS:

1. Programming like shared library.
2. Solution for memory constrains problem.
3. Performance and scalability.

# Global Array

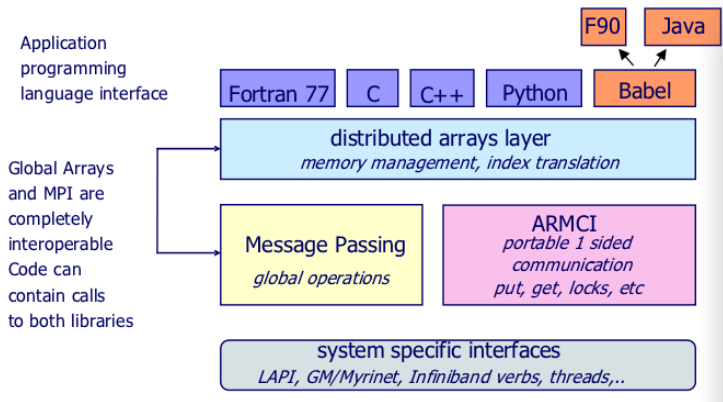
1. An instantiation of PGAS model.
2. One sided communication.
3. compatible with the Message Passing Interface



# Global Array

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #include "ga.h"
4  #include "macdecls.h"
5  int ndim,lo[1],hi[1],ld[1];
6  void test1D()
7  {
8      int dims[1]={4096*4096};
9      int chunk[1]={-1};
10     int g_a=NGA_Create(C_INT,1,dims, " A ",chunk);
11     int value=10;
12     GA_Fill(g_a,&value);
13     GA_Print_distribution(g_a);
14     lo[0]=0; hi[0]=200;
15     int buff[200];
16     NGA_Get(g_a,lo,hi,buff,ld);
17     GA_Destroy(g_a);
18 }
19 int main(int argc, char **argv)
20 {
21     MPI_Init(&argc,&argv);
22     GA_Initialize();
23     int pid=GA_Nodeid();
24     int psize=GA_Nnodes();
25     printf(" hello from process %d of %d \n ",pid,psize);
26     int i;
27     for(i=0;i<2000;i++)
28         test1D();
29     GA_Terminate();
30     MPI_Finalize();
31     return 0;
32 }
```

# Global Array Layers



# Global Array Performance

**Table 1: Latency (in microseconds) in GA and ARMCI operations**

Operation/platform	Linux 1.5GHz IA64 Elan-4	Linux 1GHz IA64 4X Infiniband	Linux 2.4GHz IA32 Myrinet-2000/C card	Linux 2.4GHz IA32 shared memory
ARMCI Get	4.54	16	17	0.162
GA Get	6.59	22	18	1.46
ARMCI Put	2.45	12	12	0.17
GA Put	4.71	16	13	1.4

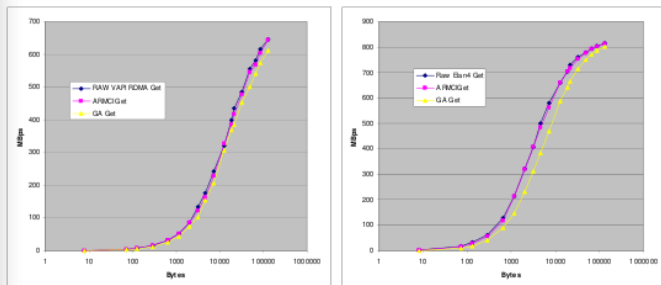


Figure 7: Comparison of GA Get with ARMCI Get and native protocols performance Left: InfiniBand (IA64), Right: Quadrics Elan4 (IA64)

# Global Array Performance

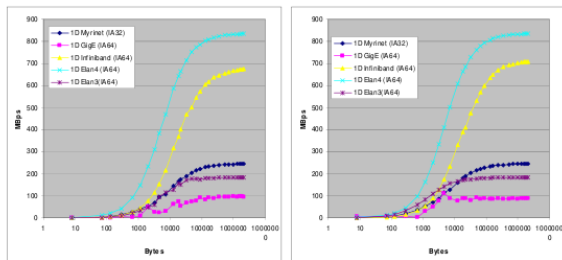


Figure 8: Performance of basic GA 1D operations on Linux clusters: get (left), put (right)

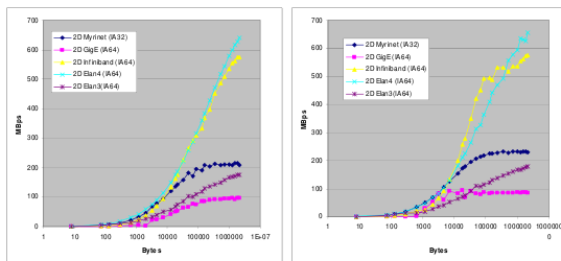
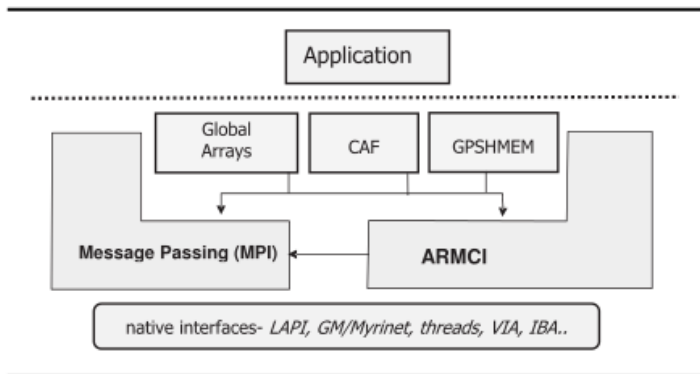


Figure 9: Performance of basic GA 2D operations: get (left), put (right)

# Global Array Advanced Features

1. Ghost Cells.
2. Nonblocking Communication.
3. compatible with the Message Passing Interface
4. Locks and Atomic Operations.
5. Disk Resident Arrays.
6. GA Processor Groups.

A Portable Remote Memory Copy library for Distributed Array Libraries and Compiler Run-time Systems



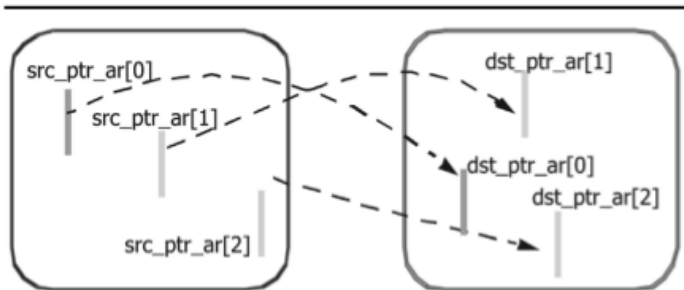
**Fig. 1 ARMCI can coexist with MPI as an implementation layer for multiple programming models and applications. Collective operations are provided by MPI.**

# ARMCI Support for noncontinuous data transfer

1. Noncontinuous data transfer are common in scientific computing.
2. Most of remote copy systems do not offer noncontiguous functionality with example like SHMEM, LAPI, Active Messages run-time system for Split C.
3. performance penalty when packing.

# ARMCI noncontinuous data transfer vector format

```
1  int ARMCI_PutV(armci_giov_t *dsrc_arr, int arr_len, int proc);
2  typedef struct {
3      void **src_ptr_ar; - Source starting addresses of each data segment.
4      void **dst_ptr_ar; - Destination starting addresses of each data segment.
5      int bytes; - The length of each segment in bytes.
6      int ptr_ar_len; - Number of data segment.
7  }armci_giov_t;
```



**Fig. 3 Source and destination pointer arrays. Non-contiguous data transfer using generalized vector format.**

# ARMCI noncontinuous data transfer strided format

```
1 int ARMCI_PutS(void* src_ptr, int src_stride_ar[], void* dst_ptr,  
2   int dst_stride_ar[], int count[], int stride_levels, int proc);  
3 src_ptr      - Source starting address of the data block to put.  
4 src_stride_arr - Source array of stride distances in bytes.  
5 dst_ptr      - Destination starting address to put data.  
6 dst_stride_ar - Destination array of stride distances in bytes.  
7 count       - Block size in each dimension. count[0] should be the  
8 number of bytes of contiguous data in leading dimension.  
9 stride_levels - The level of strides.  
10 proc       - Remote process ID (destination).
```

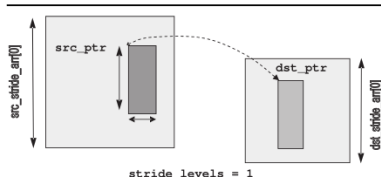


Fig. 5 Strided format for 2-dimensional array sections using column major layout.

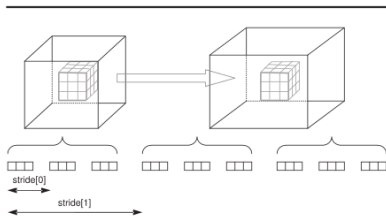


Fig. 7 Memory layout and storage requirements needed to describe transfers of a  $3 \times 3 \times 3$  section of a three-dimensional array: The vector API uses 18 pointers (source & destination for 9 segments) and  $1 + 1$  ints = 20 words. The strided API needs 2 pointers (source and destination) and  $2 + 2 + 1 + 3$  ints = 10 words.

# Memory Allocation

1. ARMCI\_Malloc: a collective memory allocator.
2. ARMCI\_Malloc\_local: just like malloc but it is shared memory.

This requirement allows ARMCI to use the type of memory that allows fastest access (e.g. shared memory on SMP clusters) which avoid some extra memory copy.

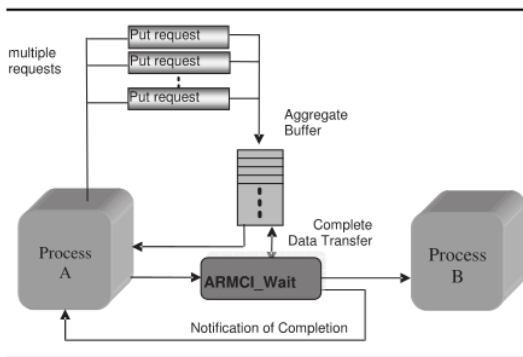
# ARMCI Nonblocking Communication

1. ARMCI\_NbPut, ARMCI\_NbGet
2. ARMCI\_NbPutV, ARMCI\_NbGetV
3. ARMCI\_NbPutS, ARMCI\_NbGetS

Nonblocking communication as a technique for latency hiding by overlapping communication with computation implicitly

# ARMCI Aggregation

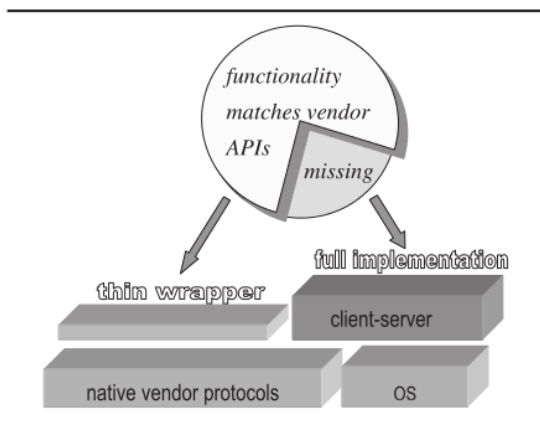
- 1) explicit aggregation: the multiple requests are combined by the user through the use of the strided or generalized I/O vector data descriptor, and
- 2) implicit aggregation: the combining of individual requests is performed by ARMCI.



**Fig. 8 Implicit aggregate data transfer.**

# High Performance Implementation

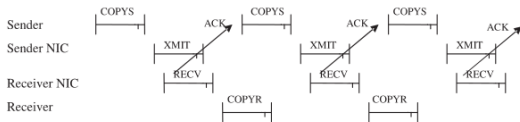
Thin-Wrapper: an ARMCI call is directly provided to it by the native communication layer.



**Fig. 9 Addressing portability in ARMCI implementations.**

# High Performance Implementation

## Pipeline armci put

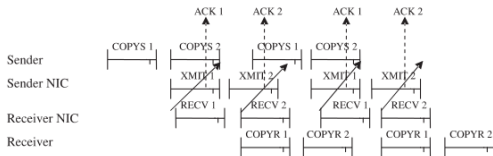


Nonpipelined version of ARMCI Put operation

COPYS - copy from source to Message Send Buffer

XMIT/RECV - The transmission and reception phases of message

COPYR - copy from the provided receive buffer to Destination



Pipelined version of ARMCI Put operation

# ARMCI Example Code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "armci.h"
4  #include <mpi.h>
5  int me,nprocs;
6  int LOOP=10;
7  int main(int argc, char **argv)
8  {
9  int k,i; double **myptrs[10];
10     MPI_Init(&argc,&argv);
11     MPI_Comm_rank(MPI_COMM_WORLD,&me);
12     MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
13     ARMCI_Init();
14     for(k=0;k<10;k++){
15         myptrs[k] = (double **)malloc(sizeof(double *)*nprocs);
16         ARMCI_Malloc((void **)myptrs[k],400000*LOOP*sizeof(double));
17         for(i=0;i<LOOP;i++)myptrs[k][me][i]=me+0.414;
18         MPI_Barrier(MPI_COMM_WORLD);
19         for(i=0;i<LOOP;i++){
20             ARMCI_Get(myptrs[k][(me+1)%nprocs]+i,myptrs[k][me]+i,sizeof(double),(me+1)%nprocs);
21         }
22         for(i=0;i<LOOP;i++){
23             armci_hdl_t nbh;
24             ARMCI_INIT_HANDLE(&nbh);
25             ARMCI_NbGet(myptrs[k][(me+1)%nprocs]+i,myptrs[k][me]+i,sizeof(double),(me+1)%nprocs,&nbh);
26             ARMCI_Wait(&nbh);
27         }
28     }
29     for(k=0;k<10;k++)ARMCI_Free(myptrs[k][me]);
30     MPI_Barrier(MPI_COMM_WORLD);
31     ARMCI_Finalize();
32     MPI_Finalize();
33 }
```

# What is X10?

1. an experimental new language currently under development at IBM.
2. a safe, modern, parallel, distributed object-oriented language, with support for high performance computation over distributed multi-dimensional arrays

## X10 DESIGN GOALS

1. Safety: illegal pointer references, type errors, initialization errors, buffer overflows are to be ruled out by design.
2. Analyzability: X10 programs are intended to be analyzable by programs (compilers, static analysis tools, program refactoring tools) and like java language.
3. Scalability: contributes to performance, and also productivity because the effort required to make an application scale can be a major drain on productivity.
4. Flexibility: exploit multiple forms of parallelism: data parallelism, task parallelism, pipelining, input parallelism etc.

# Key Design Feature

1. New programming language
2. Extend a modern OO foundation of Java.
3. PGAS.
4. Focus on dynamic asynchronous activities: lightweight “threads” can be created locally or remote.
5. Include a rich array sub-language that supports dense and sparse distributed multi-dimensional arrays.

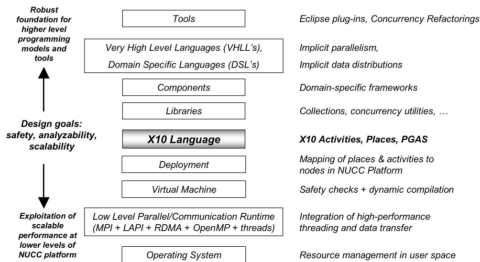


Figure 1: Position of X10 Language in Software Stack for NUCC Systems

# X10 PROGRAMMING MODEL: PLACES

1. Place is a collection of resident (non-migrating) mutable data objects and the activities that operate on the data.
2. Every X10 activity runs in a place.
3. the set of places are ordered and their methods next() and prev().

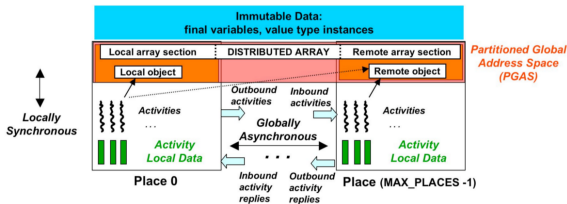


Figure 2: Overview of X10 Activities, Places and Partitioned Global Address Space (PGAS)

# X10 PROGRAMMING MODEL: Asynchronous Activities

```
1  final place Other = here.next();
2  final T t = new T();
3  finish async (Other){
4      final T t1 = new T();
5      async (t) t.val = t1;
6  }
```

1. `async (P) S`: execute statement `S` at `P` place.
2. `async S` is treated as shorthand for `async(here) S`, where `here` is a constant that stands for the place at which the activity is executing.
3. Any method-local variable that is accessed by more than one activity must be declared as `final`.
4. The execution of a statement by an activity is said to terminate locally when the activity has finished all the computation related to that statement.

# X10 PROGRAMMING MODEL: Asynchronous Activities

Activity Termination, Rooted Exceptions, and Finish.

1. A statement is said to terminate globally when it has terminated locally and all activities that it may have spawned (if any) have, recursively, terminated globally.
2. A statement terminates abruptly when it throws an exception that is not handled within its scope; otherwise it terminates normally. Parent thread may finish when child thread terminate abruptly.
3. The statement finish  $S$  in X10 converts global termination to local termination.
4. There is an implicit finish statement surrounding the main program in an X10 application.

## X10 PROGRAMMING MODEL: other features

1. multidimensional array support: `int value [.] A = new int value[[1 : 10, 1 : 10]] (point[i,j]) { return i+j; };`
2. *for* for sequential iteration by a single activity:  
`for ( point p : A ) A[p] = f(B[p]) ;`
3. *foreach* for parallel iteration in a single place.
4. *ateach* for parallel iteration across multiple places.
5. X10 Clocks as functionality like barrier but it supports advanced features like varying sets of activities and deadlock-free.
6. Support atomic block.

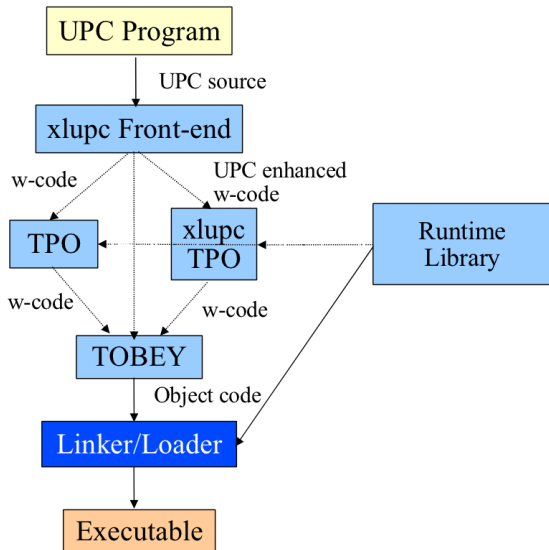
# Introduction to UPC

1. Parallel extension to C – Programmer specifies shared data – Programmer specifies shared data distribution (block-cyclic) – Parallel loops (`upc_forall`) are used to distribute loop iterations across all processors
2. Data can be private, shared local and shared remote data – Synchronization primitives: barriers, fences, and locks – Latency to access local shared data is typically lower than latency to access remote shared data
3. Flat threading model – all threads execute in SPMD style

## Three main components

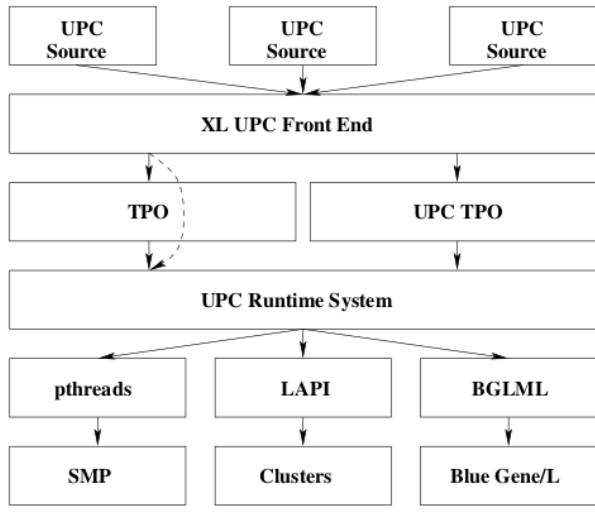
1. Front End(FE): parses different languages into a common intermediate representation(W-Code).
2. TPO(Toronto Portable Optimizer): a high-level optimizer that performs machine-independent compile-time and link-time optimizations.
3. A code generator(TOBEY): performs machine-dependent optimizations and generates code appropriate for the target machine.

# xlupc Compiler Architecture



- ▶ Designed for scalability
- ▶ Implementations available
  1. SMP using pthreads.
  2. clusters of workstations based on the Low-level Application Programming Interface (LAPI) library.
  3. BlueGene/L using the BlueGene/L message layer.

# XL UPC Compiler and Runtime System



**Figure 1.** XL UPC Compiler and Runtime System

# Shared Variable Directory in a PGAS distributed-memory machine

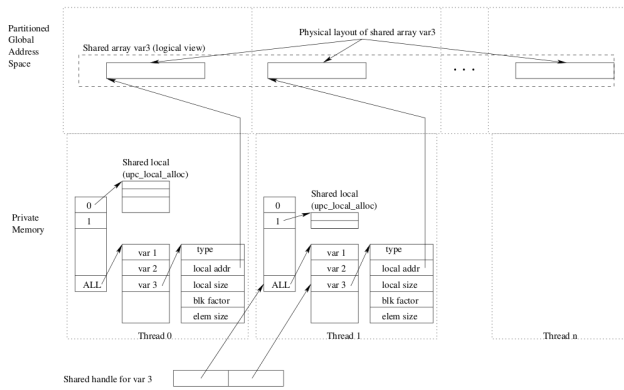


Figure 2. Shared Variable Directory in a PGAS distributed-memory machine.

# Compiler Optimization

```
OPTIMIZE SHARED ARRAY INDEX(Procedure p)
1. for each loop  $L_i$  in p
2.   if  $L_i$  is not a upc_forall loop
3.     continue
4.   endif
5.   for each shared memory reference  $R_s$  is  $L_i$  do
6.     if DIST_MEM_ARCH and  $R_s$  is non-local
7.       continue
8.     endif
9.      $R_{handle} \leftarrow$  SVD handle for  $R_s$ 
10.     $L_i^{Preheader}.Add(R_{address} \leftarrow \text{BaseAddress}(R_{handle}))$ 
11.     $off \leftarrow elt\_sz * ((blk\_sz * course) + phase)$ 
12.    if  $R_s$  is a def
13.       $sym\_data \leftarrow$  data to store to  $R_s$ 
14.      if  $R_s.DataType$  is intrinsic
15.         $L_i^{Body}.Add(\text{store}_{ind}(R_{address}, off, data))$ 
16.      else
17.         $L_i^{Body}.Add(\text{memcpy}(R_{address} + off, data, elt\_sz))$ 
18.      endif
19.    else
20.       $sym\_dst \leftarrow$  location to store data from  $R_s$ 
21.      if  $R_s.DataType$  is intrinsic
22.         $L_i^{Body}.Add(dst \leftarrow \text{load}_{ind}(R_{address}, off))$ 
23.      else
24.         $L_i^{Body}.Add(\text{memcpy}(dst, R_{address} + off, elt\_sz))$ 
25.      endif
26.    endif
27.     $L_i^{Body}.Remove(R_s)$ 
28.  endfor
29. endfor
```

Figure 3. Optimizing Shared Array Indexes (Local Memory).

`DIST_MEM_ARCH` is a flag indicating that the target architecture

# Evaluation: Random Access Benchmark and EP STREAM Benchmark

Threads	Performance	Memory TBytes		efficiency
	(GUPS)	used	total	(%)
1	5.4E-4	0.000128	0.000512	100
2	7.8E-4	0.000256	0.000512	72
4	1.3E-3	0.000512	0.001	61
64	0.02	0.008192	0.016	61
2048	0.56	0.250000	0.500	51
4096	1.11	0.500000	1.000	50
8192	1.70	1.000000	2.000	38
16384	3.36	2.000000	4.000	38
32768	6.10	4.000000	8.000	34
65536	11.54	8.000000	16.000	33
131072	16.72	8.000000	16.000	23

**Table 1.** Random Access performance results.

Threads	Performance	Memory	efficiency
	(GB/s)	TBytes	(%)
1	0.73	0.000128	100
2	1.46	0.000256	100
4	2.92	0.000512	100
64	46.72	0.008192	100
65536	47827.00	8.000000	100
131072	95660.77	8.000000	100

**Table 2.** STREAM Triad performance results.