An Intermediate Library for Multi-GPUs Computing Skeletal

Le Duc Tung*, Nguyen Huu Duc*†, Pham Tuan Anh *†,
Ngo Huy Hoang*, and Nguyen Minh Thap*

*High Performance Computing Center
Hanoi University of Science and Technology, Vietnam
tungld-hpcc@mail.hut.edu.vn, {hoang.ngo.h, towernguyenminh}@gmail.com

†School of Information and Communication Technology
Hanoi University of Science and Technology, Vietnam
{anhtpt, duch}@soict.hut.edu.vn

Abstract—This paper introduces a library which supports programmers to write parallel programs on GPU architecture, especially with a system consisting of multi-GPUs. The library is designed from the idea of skeletons, which helps us to make parallel programs easily and quickly as if writing sequential programs. Skeletons usually are described by functional language which supports high-order function totally. Because of the performance and popularity of C++ language, we try to re-annotate C++ language to support high-order functions completely, hence, it is convenience for us to create general-purpose skeletons.

Index Terms—Multi-GPUs, parallel programming, Skeleton.

I. INTRODUCTION

In the trend toward many-core architecture, NVIDIA introduced Compute Unified Device Architecture (CUDA), a multi-processing architecture for GPUs, which includes both hardware and software solutions. GPUs are now powerful programmable machines with computing capacity up to hundred times higher than normal CPUs. The power of GPUs inspires us to design and implement a skeletal library for GPU systems.

The power of skeletons will be utilized if their implementations support high-order function which accepts a function as input and returns another function. The input function is usually defined by users when they write their program, this makes the program flexibility. Although CUDA framework uses C language as fundamental language, users still have to remember many kind of modified keywords for specific functions running on GPU. Therefore, supporting the user-defined functions which seamlessly executes on GPU is an important feature.

The remain of our paper is organized into four sections: the first one introduces some related-works which are the approach of skeletons for programming and some brief information on CUDA architecture. Our main contribution will be showed in the second section in which consists of designation and implementation of the library. In the third part, we do experiments with two basic problems to see the performance and convenience of our library. The final section is our conclusion.

In our paper, we use the notation of Haskell to represent skeletons. The symbol $\oplus$ denotes an operator which is both associative and communicative. "$f \ x"$ and "$f \ x \ y$" means applying a function $f$ to one or two arguments respectively. The notation $f :: a \rightarrow b$ denotes the type of function $f$, where function $f$ accepts an input with a type of $a$ and returns an output with a type of $b$. $g \circ f$ denotes the composition of two functions $f$ and $g$, the output of $f$ is the input of $g$.

II. RELATED WORK

A. CUDA Architecture

CUDA is a unified processing architecture including both hardware and software designed for parallel computing. CUDA enables NVIDIA’s modern GPU to execute programs implemented in C, C++, FORTRAN and many other languages. These programming languages have one thing in common: they are designed for sequential executing model; CUDA maintains this model and extends it with a minimal set of abstract objects describing the parallel portion of the implementation. This way, programmers can focus more on designing parallel algorithm instead of wasting efforts on implementing mechanisms for controlling concurrency.

A CUDA program consists of two parts: the host part runs on CPU(s) and the other contains one or many kernels and device functions running on GPU(s). When a CUDA program is executed, the host part can invoke a parallel kernel on the device. The kernel will then be concurrently executed N times by N different CUDA threads. For convenience, these threads can be organized into a thread hierarchy. Threads can be groups into one-dimensional, two-dimensional, or three-dimensional thread blocks which can then be organized into a one-dimensional or two-dimensional grid of thread blocks.

B. The approach of skeleton for parallel programming

Skeletal parallel programming [5], [6] has been known as a promising approach for parallel programming. In this approach, programmers build their program based on pre-defined functions (skeletons), this components help parallelism become easier, and hide many factors related to lower-level parallel implementation on different architectures. There are
many definitions about skeleton, but in general, it is like patterns in software engineering with clear definition for its interface. The source code of a skeletal parallel program looks very simple and easy to understand, it is almost the same as that of a sequential program.

Consider a simple skeletal program called Divide-Conquer skeleton. This skeleton applies for a class of data-parallelism problems. Its definition is as follows:

\[
D\_C \text{ indivisible split join } f = F
\]

\[
\text{where } F\_P = f\_P, \text{ if indivisible } P
\]

\[
= \text{ join } (\text{map } F (\text{split } P)), \text{ otherwise}
\]

The skeleton consists of three functions, function `indivisible` checks the indivisible of input data, function `split` is to calculate and divide the input data equally and function `join` to combine results returned from computation. However, these functions is automatically running by the system. The burden to users is just to define a function `f` to apply for each element `P` in the input data. `f` is usually called user-defined function.

Some libraries provide divide-conquer skeletons can be listed as SkeTo [2], SkePU [7] for C++ language, Skandium for Java language [1].

Besides, there are some other kind of skeletons such as: iterative combination skeleton, cluster skeleton, task queue skeleton [5].

III. MultiSkel Library

A. Architecture

The library is built on CUDA framework and pthreads library with the architecture shown in figure 1. The library plays a role of abstract level. Therefore, users do not need to pay attention on details of the model of architecture or implementation of CUDA, or how to make multi-threads to control multi-GPUs.

Because of the limit of function pointer in CUDA architecture, we use a source-to-source compiler from Clang library [3]. The compiler will convert function pointers in skeletons from user’s source code into function calls inside the body of the function. Then, function calls will be annotated to generate CUDA-compatible code. Using a such compiler helps to increase the scalability of implementation environment for MultiSkel library. For example, if we would like to support more computation power by using multi-CPUs, we just modify source code generated by the compiler without changing the programming interface. After processing, user’s program will be compiled by nvcc or gcc to run on both GPU and CPU. The figure 2 shows the important role of source-to-source compiler in our library.

APIs in the library are categorized into three groups as in the figure 3:

- **Environment management**: These APIs are called when users want to start or finish computing on GPUs.
- **Memory management**: Users call these functions when they want to perform operations relating to memory on GPUs.
- **Skeletons**: This is the main component of the library including skeletons for parallel computation using GPUs.

To support multi-thread feature for functions of memory management and skeletons, it can be use one of two models: fork-join and thread-pool. With the model of thread-pool, when there is no jobs to execute, threads are in sleep status. When a job comes, system will send a message to wake up threads, and threads will process assigned jobs. The advantage of that model is threads are created once. With the model of fork-join, after finishing assigned jobs, threads end and are destroyed to release memory. Threads are re-created after each execution.

Compared to thread-pool model, fork-join model has lower performance due to cost of creation of threads. However, implementing thread-pool model in the context of skeletons is nontrivial. Because threads are only created once, we can not use template feature of C++ language to

![Fig. 1. The architecture of MultiSkel library.](image)

![Fig. 2. Role of source-to-source compiler](image)

![Fig. 3. Main components in MultiSkel library.](image)
implement skeletons. Different skeletons have different user-defined functions, or, in other word, types of input data for each of skeletons are not the same. Meanwhile, each threads at the time of its creation is sticked to a specific function or data. Therefore, it is not easy to use a common thread for many skeletons. In our paper, our approach is to use fork-join model.

When a function of memory management and a skeleton is called, the library will generate \( N \) sub-threads correspondent with \( N \) GPUs in the system. While sub-threads do its job, the main thread is just waiting for returned results from each sub-thread. This process is shown in the figure 4.

![fork-join model for multi-threads](image)

Fig. 4. fork-join model for multi-threads.

B. Designation of APIs and Skeletons

Because functions of environment are single threads [Table I], we just mention multi-threads APIs. With multi-threads APIs, their implementing structure consists of three parts as in the figure 5.

![Pseudo-code to implement multi-threads APIs](image)

Fig. 5. Pseudo-code to implement multi-threads APIs.

1) Function MultiSkel_api_function: user will call this function directly when they want to use MultiSkel. This function are implemented by the main thread which have the role of creation of sub-threads. After that, the main thread will wait for the completion of sub-threads.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuInit()</td>
<td>Initializes parameters for environment and global variables. cuInit() must be called before every operation on GPU.</td>
</tr>
<tr>
<td>cuFinalize()</td>
<td>Releases global variables.</td>
</tr>
</tbody>
</table>

TABLE I
FUNCTIONS OF ENVIRONMENT MANAGEMENT.

2) Function thread_code: this function controls a sub-thread to utilize a specific GPU in the system. Based on initial values from MultiSkel_api_function, this function will initialize its own values for its thread, and set the area of data needed for computation.

3) Function kernel_function: this function actually execute parallel computation using GPU. In the running time, this function only call functions which are able to execute on GPU, not functions running on CPU or host.

Because there is only one argument for thread_code function, we need an suitable data structure to store input data. The data structure should include three main pieces of information as follows:

- thread_code: this value is unique for each thread and be used to distinguish different threads. This value also is used for determining input data for a thread.
- data: includes both input data and output data. This variable is usually a pointer.
- size_of_data: bases on thread_code, a pointer to the first element of data array, and size of data, it is easy to find out the data offset for each thread. This will be mentioned in III-B2

1) Environment Management: MultiSkel library provides two functions to manage implementation environment: cuInit() and cuFinalize(). Being different with the other APIs, the implementation of these functions do not use pthreads library. The table I lists the details of these functions.

Before operating with memory on GPU or skeletons, programmer must call the function cuInit() to initialize variables needed for the execution of the program on GPU. Firstly, cuInit() determines the number of GPU in the system and the number of thread to create through CUDA library. For each thread, the system will initialize an array data with the type of std::vector to store the location of memory allocated. To map a pointer with actual memory on GPU, A global variable memTLU with the type of std::map is used. For convenience, a variable memCounter is used to index memory, initially, its value is zero. The details of these variables will be discuss in III-B2.

The work of cuFinalize() is in contrast with the one of cuInit(). cuFinalize() releases variables initialized by cuInit() such as memTLU and data.

2) Memory Management: To manage memory on GPUs, MultiSkel provides four APIs as follows:

- cuAlloc(): this API is used to allocate memory.
• cuFree(): this API is used to release allocated memory.
• cuGetVector(): this API is used to transfer data from GPU to main memory on host (CPU).
• cuSetVector(): this API is used to transfer data from main memory to GPU.

Note that, these functions only use to operate with memory on GPU. With RAM memory, programmer still uses functions malloc, free, or operators like new and delete as normal. MultiSkel library allows to execute parallel computation with many GPU concurrently, therefore data for computation should be distributed on many GPUs. A mechanism which hides the distribution of data with users is very important.

Management of memory in MultiSkel is based on an array called std::vector to store pointers to memory block on GPU. \( i \)\(^{th} \) element in the array stores information of memory of \( j \)\(^{th} \) GPU in the system. When a thread accesses the memory on GPUs, it uses thread_code to determine an element in the array. Using both data and memTLU, we can map a user pointer to many pointers to manage data on multi-GPUs. Value of memTLU is the address of a memory cell, this ensures the value is unique. A global variable memCounter is used to number for area of memory allocated.

To allocate a memory area on multi-GPUs, MultiSkel does as follows:

1) Allocates memory for an element with the type defined by user, and a pointer a to this area. Then, user will use pointer a to refer to actual memory on GPUs.
2) \( \text{memTLU}[a] = \text{memCounter}++ \); stores key a in memTLU with the value of memCounter, then increases memCounter by 1.
3) Initialises main thread and creates sub-threads.
4) Sub-thread calculates its memory area. With an array of \( N \) elements, the calculation will be as follows:
   
   
   \[
   \text{size} = \frac{\text{numOfElement}}{\text{numOfGPU}} \quad \text{if} \quad \text{(threadID} = \text{(numOfGPU} - 1)\} \quad \text{size} = \frac{\text{numOfElement} - (\text{size} \ast \text{threadID})}
   \]

5) Sub-thread actually allocates its memory area on GPU managed by that sub-thread, and stores the address of the memory area in the array data

Compared to allocation operation, operation for releasing memory is simpler.

1) \( \text{memID} = \text{memTLU}[a] \); Sub-threads get the index of memory area pointed by a.
2) \( \text{aOnGPU} = \text{data}[\text{threadID}][\text{memID}] \); Based on the index of memory area, get a pointer in data[threadID].
3) Sub-threads do release memory on GPU using CUDA library.

With functions for transferring data, in addition to get actual address of memory block and calculate size of memory on each GPU, threads have to calculate offset of memory on the main memory (memory on host). The formula is as follows:

\[
\text{offset} = \text{size} \ast \text{threadID}
\]

Sub-threads transfer the area of data with size elements at the address \( a + \text{offset} \) on the main memory to GPUs to manage it.

C. Parallel Skeletons

MultiSkel library consists of six skeletons for parallel computation on multi-GPUs. map, reduce and scan are primitive skeletons. zipWith, mapReduce, zipWithReduce are advanced skeletons optimized for some class of specific problems.

1) map skeleton: Map is a skeleton derived from functional language. Map allows to apply a function concurrently on every element in a list. The formal definition of Map is as follows:

\[
(\text{map } f) : [a] \rightarrow [b]
\]

where, \( f \) is a function with the type of \( f : a \rightarrow b \).

In MultiSkel library, map skeleton is implemented as function cuMap with the declaration as follows:

\[
\text{template <class T1, class T2> void cuMap(T2 (*f)(T1), int N, T1 *in, T2 *out)}
\]

Using the template feature of C++ language, the skeleton accepts the arguments of any type. The skeleton map is implemented using the structure mentioned before in the figure 5:

- **cuMap** - the API function
- **mapWorker** - the thread function
- **mapKernel** - the core

Besides the function \( f \), the other arguments of cuMap will be assigned to corresponding fields in the structure map_info which is passed to thread mapWorker. Based on the fields of the structure map_info, thread mapWorker determines the size of data to process, resolves address space pointed by pointers in and out, and calls kernel mapKernel corresponding to the function \( f \).

Implementing the skeleton map is fairly simple as shown in figure 6. Threads running on GPU compute its identification numbers using the built-in variables in CUDA such as threadIdx, blockIdx and blockDim and then process its data element. After processing an element, the threads jump to the next element with the step of \( s \) that is the total number of threads running on the GPU. This process is repeated until all elements of the input array are treated.

2) reduce skeleton: The skeleton Reduce is primitive and important in the parallel data. The operation takes a binary operator which is associative \( \oplus \) and an array of \( n \) elements

\[
[a_0, a_1, \ldots, a_{n-1}]
\]

and returns a scalar value

\[
a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}
\]
template <class T1, class T2>
__global__ void mapKernel(int n, T1* in, T2* out)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < n) {
        out[tid] = f(a[tid]);
        tid += blockDim.x * gridDim.x;
    }
}

Fig. 6. The core of skeleton map

Corresponding to the above definition, the prototype of the function cuReduce, the function that implements the skeleton reduce, is

template <class T1>
void cuReduce(T1 (*f)(T1, T1), int N, T1 *in, T1 *out)

3) scan skeleton: The skeleton scan is a good example of the sequential calculations, but there is an efficient parallel algorithm in [4]. The inclusive scan takes an associative binary operator ⊕ and an array of n elements

\[ [a_0, a_1, \ldots, a_{n-1}] \]

and returns

\[ [a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + \ldots + a_{n-1})] \]

4) zipWith skeleton: zipWith is not an essential skeleton but very efficient with applications that deals with two arrays with the same type. Skeleton zipWith takes a function \( f \) and two arrays of \( n \) elements

\[ [a_0, a_1, \ldots, a_n], [b_0, b_1, \ldots, b_{n-1}] \]

and returns another array

\[ [f(a_0, b_0), f(a_1, b_1), \ldots, f(a_{n-1}, b_{n-1})] \]

Similar to map skeleton, this skeleton do not require communication, hence, this skeleton is a linear function.

5) zipWithReduce and mapreduce skeleton: MapReduce and zipWithReduce skeletons are complex skeletons. MapReduce is a composition of map and reduce, it takes a function \( f \), an associative binary operator \( \oplus \) and an array of \( n \) elements

\[ [a_0, a_1, \ldots, a_n] \]

and returns

\[ f(a_0) \oplus f(a_1) \oplus \ldots \oplus f(a_{n-1}) \]

zipWithReduce is a composition of zipWith and reduce, it takes a function \( f \), an associative binary operation \( \oplus \), and two arrays of \( n \) elements

\[ [a_0, a_1, \ldots, a_n], [b_0, b_1, \ldots, b_{n-1}] \]

and returns a scalar value of

\[ f(a_0, b_0) \oplus f(a_1, b_1) \oplus \ldots \oplus f(a_{n-1}, b_{n-1}) \]

Compared to the skeletons map, reduce, zipWith, the two skeletons are more effective composition due to no intermediate arrays required. Instead of making two calls to two skeletons, do a complex skeleton. The programmer is encouraged to use the complex skeletons whenever possible.

IV. Experiment

The experiments are performed at High Performance Computing Center with the configuration as follows:

- One processor AMD Athlon X4 620 2.6GHz Quad-Core
- 8GB RAM
- Two NVIDIA GeForce GTX 295 with 4 GPUs, each GPU has 240 cores @ 1242MHz and 896MB GDDR3 @ 999MHz.

A. Scalar product

The scalar product of two vectors \( a = [a_1, a_2, \ldots, a_n] \) and \( b = [b_1, b_2, \ldots, b_n] \) defined as:

\[ a \cdot b = \sum_{i=1}^{n} a_i \times b_i = a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n \]

where \( n \) is the size of the vector.

To perform the operation \( a_i \times b_i \), we use the skeleton zipWith with the function of multiplication

\[ \text{zipWith}(\times, a, b) = [a_1 \times b_1, a_2 \times b_2, \ldots, a_n \times b_n] \]

Transmitting the result of skeleton \( \text{zipWith} \) to skeleton reduce with the operator \( + \), we use the scalar product of two vectors \( a \) and \( b \).

\[ a \cdot b = \text{reduce}(+, \text{zipWith}(\times, a, b)) = \text{reduce}(+, [a_1 \times b_1, a_2 \times b_2, \ldots, a_n \times b_n]) = a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n \]

We may also use the complex skeleton \( \text{zipWithReduce} \) with the function of multiplication and the operator \( + \).

\[ a \cdot b = \text{zipWithReduce}(\times, +, a, b) \]

Since the scalar product is an operation that is used frequently, it is put in many libraries. In this experiment, to achieve an objective evaluation, we compare the function of the scalar product using the library MultiSkel with the library CUBLAS which is an implementation of BLAS (Basic Linear Algebra Subprogram) developed by NVIDIA CUDA.

The function of the scalar product is experimented with two arrays of \( 2^{24} \) to \( 2^{28} \) floating point numbers. The experimental results are presented in table II.

Again, the programs running on GPU is 50 to 200 times faster than on the CPU. It is remarkable that a program on a GPU using a skeleton version of MultiSkel runs 10 % faster than the version performed in the library CUBLAS. This result demonstrates, to some extent, the quality of the library
MultiSkel. We should also notice that the versions of complex skeleton run up 2 times faster than the one of two primitive skeletons.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>1GPU cuBLAS</th>
<th>1GPU 1 skeleton</th>
<th>4GPU 2 skeletons</th>
<th>4GPU 1 skeleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{24}$</td>
<td>89,020</td>
<td>1,408</td>
<td>2,915</td>
<td>1,333</td>
<td>1,458</td>
</tr>
<tr>
<td>$2^{25}$</td>
<td>177,657</td>
<td>2,946</td>
<td>5,721</td>
<td>2,645</td>
<td>1,882</td>
</tr>
<tr>
<td>$2^{26}$</td>
<td>354,700</td>
<td>6,299</td>
<td>11,562</td>
<td>5,505</td>
<td>3,173</td>
</tr>
<tr>
<td>$2^{27}$</td>
<td>710,407</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>6,355</td>
</tr>
<tr>
<td>$2^{28}$</td>
<td>1420,532</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>12,937</td>
</tr>
</tbody>
</table>

**TABLE II**

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>1GPU cuBLAS</th>
<th>1GPU 1 skeleton</th>
<th>4GPU 2 skeletons</th>
<th>4GPU 1 skeleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{24}$</td>
<td>68,648</td>
<td>5,290</td>
<td>4,365</td>
<td>5,230</td>
<td></td>
</tr>
<tr>
<td>$2^{25}$</td>
<td>136,874</td>
<td>10,789</td>
<td>8,025</td>
<td>6,208</td>
<td></td>
</tr>
<tr>
<td>$2^{26}$</td>
<td>270,313</td>
<td>22,834</td>
<td>17,638</td>
<td>8,356</td>
<td></td>
</tr>
<tr>
<td>$2^{27}$</td>
<td>552,128</td>
<td>X</td>
<td>X</td>
<td>13,429</td>
<td></td>
</tr>
<tr>
<td>$2^{28}$</td>
<td>1089,452</td>
<td>X</td>
<td>X</td>
<td>24,713</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III**

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU</th>
<th>1GPU cuBLAS</th>
<th>1GPU 1 skeleton</th>
<th>4GPU 2 skeletons</th>
<th>4GPU 1 skeleton</th>
</tr>
</thead>
</table>

B. Pearson coefficient of correlation

In statistics, the Pearson coefficient of correlation is a measure by the linear dependence between two variables $X$ and $Y$. The mathematical expression of the Pearson coefficient of correlation is as follows:

$$ r = \frac{n \sum XY - \sum X \sum Y}{\sqrt{(n \sum X^2 - (\sum X)^2)(n \sum Y^2 - (\sum Y)^2)}} $$

The Pearson coefficient of correlation is calculated using skeletons as follows:

- $\text{sum}1=\text{reduce}(+, [x_1, x_2, \ldots, x_n])$
- $\text{sum}2=\text{reduce}(+, [y_1, y_2, \ldots, y_n])$
- $\text{sumpr}=\text{zipWithReduce}(\times, +, [x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n])$
- $\text{sumsq}1=\text{mapReduce}(<\text{sqr}, +, [x_1, x_2, \ldots, x_n]>)$
- $\text{sumsq}2=\text{mapReduce}(<\text{sqr}, +, [y_1, y_2, \ldots, y_n]>)$

$$ r = \frac{n \times \text{sumpr} - \text{sum}1 \times \text{sum}2}{\sqrt{(n \times \text{sumsq}1 - \text{sum}1^2)(n \times \text{sumsq}2 - \text{sum}2^2)}} $$

where $\text{sqr}(x) = x^2$

In this experiment, we calculate the coefficient of correlation of two matrices of $2^{24}$ to $2^{28}$ floating point numbers. The experimental results are presented in table III.

The gap between the execution time of the program for the CPU and the program for GPU is reduced. Even the experimented program of 4 GPUs is only 44 times faster than the sequential program. This happens because the number of arithmetic operations in the loop of this program is five times as much as one with the program calculating the scalar product of CPU. For programs running on the GPU, the ratio of arithmetic operations and the operations accessing the global memory does not change. With the program for multiple GPUs, it needs to initialize and run multiple thread groups, which is quite expensive. In comparison to the program carried out by using the library CUBLAS, the skeleton program is still 20% faster.

V. Conclusions

In hope of easing programming on GPU, this paper introduces a library of high-performance algorithmic skeleton with a simple programming interface. The library contains two main components: skeletons for the 1-D array data structure and a source-to-source compiler. Six basic algorithmic skeletons have been implemented. They accept all types of data and can run in parallel way on multiple GPUs. The memory management and allocation in tasks are performed automatically and transparently to the programmer. The library obviously can not exceed an experienced developer, but the results of experiments shows that the library has good quality and high performance compared to other libraries and sequential programs.

At present, the skeletons in the library MultiSkel can only treat the data structure 1-D array using the multi-GPUs architecture. In the future, we will develop the skeletons for more complicated data structure such as 2-D array or tree. Supporting for new platforms such as a multi-CPU or a GPU-cluster is also a potential research.

**REFERENCES**