# Efficient and Fault Tolerant Computation of Partially Idempotent Tasks

Edson Borin (IC/UNICAMP), Ian L. Rodrigues* (CEPETRO/UNICAMP), Alber T. Novo (CEPETRO/UNICAMP), João D. Sacramento (CEPETRO/UNICAMP), Mauricio Breternitz (AMD Research), and Martin Tygel (IMECC/UNICAMP)



## Abstract

**The popularization of multi-core processors on computational systems and cloud services enabled access to high performance computing infrastructure. However, programming for these systems might be cumbersome due to frequent system failures when using thousands of machines, poor load balancing and task scheduling. To solve these problems, we introduce a concept of partially idempotent tasks and discuss how their properties ease the implementation of fault tolerant mechanisms, load balancing on heterogeneous systems and dynamic provisioning of resources. Additionally, we propose a programming model, a scalable system and an API which executes these kind of tasks and we present an usage example of our system to a simple problem.**

## Introduction

The rising of multi-core processors of general purpose and specialized ones, like GPUs, made the development of parallel code for shared memory machines an essential skill to extract maximum performance from those hardwares. Furthermore, the popularization of computer clusters and *utility computing*, term usually used for cloud systems, enabled many people to have access to high performance computing infrastructures, thus rising the importance and demand for development of parallel code on *distributed* memory systems.

The need for parallelized code motivated the development of many programming models and interfaces in the past years. OpenMP (Dagum and Menon, 1998), OpenCL (Stone et al., 2010), CUDA (Nickolls et al., 2008), TBB (Reinders, 2007), and MPI (Forum, 1994) are some examples of application programming interfaces (API), programming models and libraries supporting parallel computational systems. However, the majority of these models or APIs requires the use of specialized compilers, operating systems, tools or libraries; or are restricted to a system with either shared memory or distributed memory. Besides that, many of these are not transparent when dealing with fault tolerant mechanisms, an essential feature to scale the execution of the program to hundreds (or thousands) of processing nodes.

Many problems like ray tracing in computer graphics, matrix multiplication, Monte Carlo integration, the CRS (Jäger et al., 2001) method for seismic analysis, belong to a class of problems called “embarrassingly parallel”. These problems are usually trivially separated into smaller, independent subproblems which, in turn can be solved in parallel with little to no restriction in the execution order. Although these problems are trivially parallelized, building a solution able to scale to hundreds or thousands of computing nodes is still a challenge in itself.

Many of the algorithms for embarrassingly parallel problems can be decoupled into *partially idempotent tasks*, or PITs. In addition to being easily parallelized, PITs can be executed multiple times without affecting the final result. We discuss the properties of PITs, propose a programming model and a scalable, fault tolerant system to compute PIT tasks called Scalable Partially Idempotent Tasks System (SPITS).

### Related Work

Recently, de Kruijf and Sankaralingam (2011) proposed an architecture of idempotent processors and demonstrated that the called “idempotent processing” allows the implementation of speculative optimization in the processor without needing costly mechanisms of check pointing for the recovery of the architectural state. The authors observed that the precise state their architecture can be rebuilt by simply re-executing idempotent regions. In another work, de Kruijf et al. (2012) implemented a compilation technique aiming the division of the program in idempotent regions. In this work, the idempotence concept was extrapolated to tasks and the idempotence property was used to facilitate the implementation of fault tolerant mechanisms and load balancing in heterogeneous systems.

The MapReduce (Dean and Ghemawat, 2008) technique splits the input data into several small parts. Each part is inputed, in parallel, to a user defined function called *map*, which outputs a key-value pair. After all key-value pair were created, MapReduce enters shuffle phase, where values with the same key are grouped into one worker node. When all values with the same key were grouped, the user defined function *reduce* is called to generate the final key-value pair output. This functionality is inside of the open source Hadoop system.

To tolerate the discrepancy of performance of processing node in heterogeneous computational systems, Hadoop execution system (called tasktracker) monitors the task progress and may start two or more identical tasks in different nodes. The first task to finish writes its results in the file system, whilst the other tasks are finishing. LATE (Longest approximate time to end) algorithm exploits the pros of the idempotent properties in commercial systems

of distributed computing, which is a part of the recent distribuition of MapReduce in Hadoop.

The SPITS extends the Hadoop approach of starting multiple copies of the same task to achieve a fault tolerant, scalable and dynamic provisioned system for partially idempotent tasks. It will be able to extend to heterogeneous systems (computational clouds, GPU, x86 and ARM).

**Partially Idempotent Tasks and its advantages**

Idempotent tasks are tasks which can be executed multiple times without altering the results after the initial execution. Figure 1 (a) shows an example of a idempotent task. Observe that, even if the program is executed multiple times, the result stored in the C matrix remains the same. On the other hand, Figure 1 (b) illustrates a non idempotent task. Note that if the code is executed twice, the result stored in matrix A will vary between executions.

```
float A[100][100];
float B[100][100];
float C[100][100];
void matrix_mul() {
 int i, j, k;
 for (i = 0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
   float tmp = 0;
   for (k = 0; k < 100; k++)
    tmp += A[i][k] * B[k][j];
   C[i][j] = tmp;
  }
 }
}
```

(a) Idempotent

```
float A[100][100];
float B[100][100];
void matrix_acc() {
 int i, j;
 for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
   A[i][j] += B[i][j];
}
```

(b) Non idempotent

Figure 1: Examples of (a) idempotent tasks and (b) non idempotent tasks.

In many cases, the consolidation of the results of all idempotent tasks is not an idempotent operation. Consider the operation defined by the concatenation of all results into one file. This operation is obviously not idempotent, because if we execute the operation twice, the file will be twice as big. Thus, we propose a concept of *partially* idempotent tasks, or PITs, where the computation of the task is subdivided in two phases: the *execution* and the *consolidation*. The execution of the task is an idempotent operation, while the consolidation is non idempotent. The focus of this paper is the efficient execution of PITs where the cost of execution is significantly higher than the consolidation of the tasks.

**Fault tolerance:** many fault tolerant systems uses the complex and costly *checkpoints* mechanisms, which allows the recovery of the system state after a failure (Egwutuoha et al., 2013). However, Kruijf et al. de Kruijf et al. (2012) observed that idempotence enables system recovery simply by re-executing the affected region when the program failed. This property allows a system which executes PITs to recover *without* checkpoints, thus avoiding expensive solutions.

Because consolidation phase is not idempotent, it cannot be re-executed to recover from an eventual failure. Thus, we propose the consolidation phase to be executed in processing nodes having other fault tolerant mechanisms. Another possibility is the execution of that phase in a small subset of nodes, reducing the probability of a failure in a system with hundreds or thousands of nodes. Both approaches can affect performance of the consolidation operation, which can be caused by expensive fault tolerant mechanisms in the first case or by reducing parallelism in the second case. But, as we stated before, the execution phase of PITs are significantly costlier than consolidation, thus reducing system load.

**Load balancing:** load balancing is a fundamental operation to maximize available resources in a parallel computational system. The *work stealing* Blumofe and Leiserson (1999) technique is a greedy algorithm which distributes work load to processing nodes on demand, and it is generally very effective when balancing loads in homogeneous systems. However, naive application of work stealing technique in heterogeneous systems might not work. Figure 2 (a) shows an example where load balancing with work stealing does not provide satisfactory performance in a heterogeneous system. In this example, the work stealing technique delegated the task $T_1$ to processor A, and task $T_2$ to processor B. However, processor A is much faster than processor B and, as illustrated in Figure 2 (b), the computation would be faster if both tasks were delegated to processor A.

Figure 2 (c) also shows an example where tasks are distributed with the working stealing technique, but by the time $T_1$ finishes in A, task $T_2$ is delegated from B processor to A, improving the system efficiency. Although in this example we have a speedup, the delegation of tasks in execution phase to other nodes might be difficult or impossible in some cases.

An alternative solution to all others is shown in Figure 2 (d). In this case, tasks are also distributed with the same technique of work stealing, however a copy of task $T_2$, which is already executing in processor B, is delegated to be executed in processor A as soon as it gets idle. By the time $T_2$ finishes in A, the system interrupts the computation of $T_2$ in B, for it is not necessary anymore. This approach improves the load balancing and avoids the whole system from getting stuck waiting for a slow node to finish its task.

The execution of the same task in multiple processing nodes could affect the final result of the computation. However, the idempotence property allows us to execute the same PIT multiple times without changing the final result. In this manner, the speculative scheduling of PITs in idle nodes enables an effective load balancing in parallel computing nodes with heterogeneous performance.

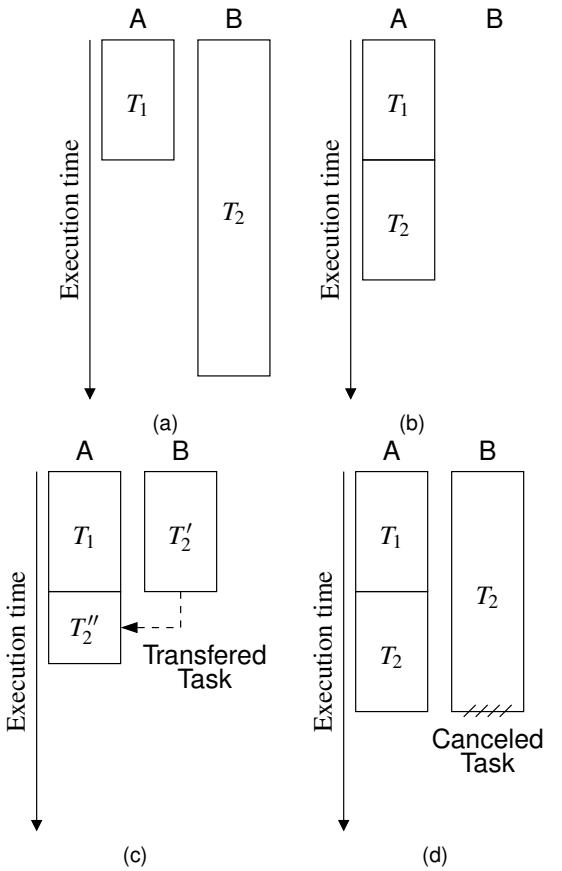


Figure 2: Example of task scheduling in heterogeneous systems.

## SPITS

SPITS is a scalable system for computing partially idempotent tasks. Following, we present the parallel programming model, the system architecture and its API.

*Parallel programming model*

The SPITS programming model consists in generating, executing and consolidating PITs. The programmer is responsible for writing four functions to fulfill those operations. The functions are:

- `generatePIT`: generates one or more partially idempotent tasks. The system calls this function successively and delegates them to processing nodes.
- `executePIT`: called on processing nodes, executes the idempotent phase of the task, producing a result. The system will transfer the results to the nodes responsible for consolidating the them.
- `commitPIT`: called once for each result generated by `executePIT` sequentially. All calls to this function are made on the same process, thus the memory is shared between executions, enabling the user to combine the tasks results. The system does not assume idempotence in this phase.
- `commitJOB`: when provided, is called after the consolidation of the task results. As this function is called on the same process which consolidated the results, the memory is shared between them.

SPITS system calls the function `generatePIT` multiple times until it returns 0, indicating that no more tasks to be generated. As tasks are generated, SPITS transmits them to processing nodes, superposing the generation, transmission and execution of tasks. This approach helps to hide the network latency.

*SPITS architecture*

SPITS architecture is a composition of four main parts: *Job Manager*, *Task Manager*, *Worker* (w) and *Committer*. Figure 3 illustrates the communication between these components.

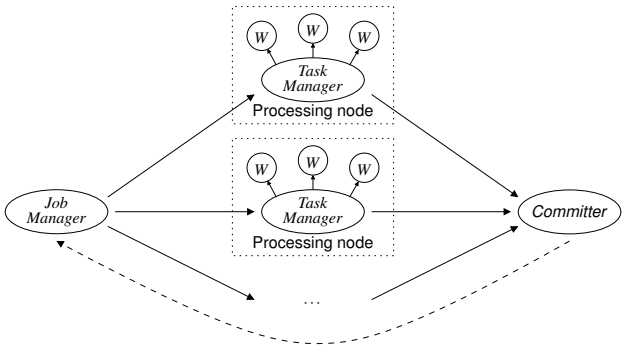


Figure 3: SPITS components.

The *Job Manager* (JM) is one of the main components of SPITS. It is responsible for generating and distributing tasks for execution in the processing nodes. Besides that, the JM is responsible for *retiring* successfully finished jobs and redistributing uncompleted tasks. As discussed in the past section, redistribution of tasks is the core of the fault tolerance mechanism and load balancing of heterogeneous systems.

The *Task Manager* (TM) is the component that executes tasks delegated to processing nodes. To accomplish that, the TM distributes tasks to *Workers*, which in turn executes the task by calling `executePIT`, provided by the user. Each worker is associated to a computing unit on the node, which can be processing cores of the CPU, GPUs, etc.

*Workers* execute the code that computes tasks and produces results as output, called *task result*. The results of the execution phase must be consolidated, however, to ease the fault tolerance mechanism (as discussed on the last section) the consolidation phase is done in a single node, so every result is transmitted to this particular node.

The *Committer* is responsible for consolidating the task results produced by the workers. To do that, *Committer* calls `commitPIT` for each received result. The fault tolerant and load balancing mechanisms will eventually replicate tasks, so the committer might receive multiple copies of a result regarding the same task. However, as consolidation is not idempotent, only one copy of the result must be consolidated. Thus, the committer is also responsible for guaranteeing to call `commitPIT` exactly once for each result.

Besides consolidating the task results, the committer also signals the job manager that a task has been successfully finalized, allowing it to retire that task, meaning it will not be eligible for re-execution by the load balancing or fault tolerance mechanisms. After consolidating all tasks, this component calls `commitJOB`, provided by the user to wrap up the computation as a whole. This function might output a sequence of bytes to be copied back to the job manager and returned to the user.

Only one instance of the job manager component is executed during the computation. In this way, the function `generatePIT` is not executed in parallel, which simplifies its implementation for not requiring data synchronization mechanisms. The same happens with the commiter component, simplifying the implementation of the `commitPIT` function. The task manager, however, can be executed in multiple processing nodes in a concurrent manner. Besides that, the task manager can handle multiple workers in the same process by calling `executePIT` in different threads, for example. For this reason, the programmer must be cautious when modifying shared data (such as global variables) on the `executePIT` function.

*Fault tolerance*

The failure of one or more processes of a parallel application could invalidate its result. Although this is not common in system with a few processing nodes, the chance of a failure occurring in systems with hundreds or thousands of nodes is high. Thus, to correctly finish a computation, the system must tolerate eventual failures.

SPITS tolerates failures in the processing nodes by using the idempotence property of the tasks while in execution phase. This property allows the system to execute this phase in multiple processing nodes redundantly. This approach allows SPITS to tolerate failures without the necessity of expensive checkpoint mechanisms.

It is important to highlight that the job manager and commiter components, responsible for task generation and consolidation, executes operations which are not idempotent, thus SPITS does not tolerate failures on the nodes executing them. However, as discussed before, the number of nodes executing these components is small (usually 1 or 2), reducing the chances of a failure in this part of the system. Besides that, the user can execute these components in processing nodes with other mechanisms of fault tolerance.

The task manager and workers components, responsible for the task execution phase, can be executed in hundreds or thousands of nodes. It is important that the system tolerates failures in nodes which execute them. In this case, the fault tolerance is done trough replication of tasks, and works in the following way: initially the job manager generates tasks and distributes to task managers. The distributed tasks are added to a WIP, or "work in progress", list where they can be re-distributed in the future. Once the result of an executing task is consolidated by the committer, the job manager *retires* that task, removing it from the WIP list. As soon as the `generatePIT` function returns zero (which indicates there are no more tasks), the job manager starts to redistribute the tasks in the WIP list in a circular manner, replicating them in the processing nodes until every task gets retired. Once all tasks are retired, the job manager sends a signal to every task manager so they can stop executing.

Processing nodes which fail will not send their results to the committer, however, the same task will certainly be send to other processing nodes, since it stays in the WIP list until it is retired. This mechanism tolerates failures of the *Fail-Stop* kind (Schlichting and Schneider, 1983). However, the system can be easily extended to tolerate byzantine failures by means of task replication and comparison of results for differences in the committer.

*Load balancing in heterogeneous systems*

The same mechanism used to tolerate failures is used to balance the system load in heterogeneous systems. As discussed before, when there are no more tasks to be generated, the job manager starts to re-distribute tasks in the WIP list until every task gets retired. As faster processing nodes finishes their tasks, the work in progress list is cycled, re-distributing those tasks to these faster nodes. As we showed in Figure 2 (d), this approach avoids waiting for slow nodes from finishing their tasks.

*Dynamic provisioning of resources*

SPITS allows processing nodes to be added or removed from the system while executing the jobs. Added nodes during computation connect with the job manager and promptly starts receiving new tasks or even re-distributed, non consolidated tasks. To remove nodes, there is no need to wait their tasks to finish, since the fault tolerance mechanism will take care of re-distributing those canceled tasks again to other, healthy nodes. In a way, SPITS allows the user or the system to dynamically adjust the resources to a job.

The flexibility provided by this mechanism simplifies the management of shared resources and dynamic provisioning of on demand resources. SPITS fits very well with the cloud computing concept, where users can easily allocate or deallocate resources whenever needed.

*C language API*

The first API developed for SPITS is dedicated to programs written in the C language. This API requires from the programmer to create a shared library exporting the following symbols[1]:

- `void* setupJM(int argc, char *argv[])`: called before the job manager starts task distribution. The return value is an address to a memory region allocated in the job manager process. This pointer is passed to the `generatePIT` function.
- `int generatePIT(void *user_data, struct byte_array *task)`: produce tasks to be executed. The tasks must be serialized in the byte sequence `task`. This sequence will be transmitted to the task managers and then to workers, where the function `executePIT` will be called. The user must return 1 in case there are more tasks to be sent, and 0 otherwise.

[1] Because of the experimental characteristic of the project, names and implementation details might change drastically. See `https://github.com/ianliu/spitz`

- `void* setupTM(int argc, char *argv[])`: called before the task manager starts executing tasks. The return value is an address pointer to a memory region allocated in the task manager process, which is passed to the `executePIT` function. Note that there will be several instances of task managers, which implies the memory is not shared between them.
- `void executePIT(void *user_data, struct byte_array *task, struct byte_array *result)`: executes the task serialized on the bytes sequence `task`. This function must write the task result in the byte sequence `result` and need to be idempotent, meaning the result cannot vary in case this function is executed more than once for the same task.
- `void* setupComm(int argc, char *argv[])`: called before the committer starts consolidating the results. The return value is a pointer to a memory region allocated on the committer process. This pointer is passed to `commitPIT` and `commitJOB` functions.
- `void commitPIT(void *user_data, struct byte_array *result)`: consolidates the results generated by the `executePIT` function. The system does not assume `commitPIT` is idempotent, thus it will be called once for each task result.
- `void commitJOB(void *setup, struct byte_array *ret)`: consolidates the work as a whole. This function is called after the all task results were consolidated with `commitPIT` function. The value written in the bytes sequence `ret` will be transmitted to the main process which started the tasks execution, allowing the result of the computation to be returned to the user.

After compiling the above functions in a shared library, the system will be capable of loading the library in execution time in each of the computing nodes, and will call them according to the node responsibility.

With these functions it is possible to execute a job in the SPITS model, taking advantage of fault tolerance and scalability. But idempotence can be restrictive for some problems, thats why we created a mechanism which enables the execution of several jobs in sequence using the SPITS model. To accomplish this another function is needed, called `spitsMain`:

- `void spitsMain(int argc, char *argv[], SpitsRunner run)`: this function is called in the job manager and receives the command line arguments and a *run* function (explained below). In the `spitsMain` function the user can program serially and, when necessary, call the *run* function to start a SPITS job.
- `void run(int argc, char *argv[], const char* dll_filename, struct byte_array* ret)`: This function starts a SPITS job and returns after all tasks were executed and consolidated. The function takes as arguments the path to a shared library and a pointer to the bytes sequence which will be written by the `commitJOB` function by the end of the computation. The system loads the library in each of the processing nodes and calls the functions `generatePIT`, `executePIT`, `commitPIT` and `commitJOB`.

The shared library mechanism decouples the program from the execution engine of SPITS tasks, allowing the same program to run in different parallel computing systems.

A primary version of the SPITS code, as well as usage examples, is available in `https://github.com/ianliu/spitz`.

*C++ language API*

The system developed also provides an interface to programs in the C++ language. In this case, the user must implement three classes: one to manage the workers, another to manage tasks and one to consolidate the tasks. These classes must extend the following classes: `SpitsJM`, `SpitsTM` and `SpitsComm`.

The class `SpitsJM` has the following virtual methods:

- `void setup(int argc, char *argv[]);`
- `bool generatePIT(Stream& task)=0;`

The class `SpitsTM` has the following virtual methods:

- `void setup(int argc, char *argv[]);`
- `void executePIT(const Stream& task, Stream& result)=0;`

The class `SpitsComm` has the following virtual methods:

- `void setup(int argc, char *argv[]);`
- `void commitPIT(const Stream& task)=0;`
- `void commitJOB(Stream& ret)`

The following listing illustrates a program in C++ using the SPITS system to calculate and approximation of the number $\pi$ using a naive Monte Carlo approach:

```
class CalcPiJM : public SpitsJM {
  int num_points;
public:
  void setup(int argc, char *argv[]) {
    num_points = atoi(argv[0]);
  }
  bool generatePIT(Stream& task) {
    if (num_points == 0) return false;
    double x = (double)rand()/RAND_MAX;
    double y = (double)rand()/RAND_MAX;
    task << x << y;
    num_points--;
    return true;
  }
};
class CalcPiTM : public SpitsTM {
public:
```

```
  void executePIT(const Stream& task,
                  Stream& result)
  {
    double x, y;
    task >> x >> y;
    result << (x*x + y*y <= 1);
  }
};
class CalcPiComm : public SpitsComm {
  int count;
  int num_points;
public:
  void setup_Comm(Stream& setup) {
    num_points = atoi(argv[0]);
    count = 0;
  }
  void commitPIT(const Stream& taskresult)
  {
    int x;
    taskresult >> x;
    count += x;
  }
  void commitJOB(Stream& ret) {
    double pi = 4.0L * count / num_points;
    cout << "Pi = " << pi << endl;
  }
};
```

## Conclusions

We introduce a concept of partially idempotent tasks and discuss how its properties ease the implementation of fault tolerance, load balancing in heterogeneous systems and dynamic provisioning of resources. Besides that we proposed SPITS, a programming model, scalable system and an API to execute these kind of tasks. A first draft implementation of the program is also provided to the community with the GPL license, with bindings for C and C++.

## Acknowledgments

We acknowledge support from the National Council for Scientific and Technological Development (CNPq-Brazil), the National Institute of Science and Technology of Petroleum Geophysics (ICTP-GP-Brazil) and the Center for Computational Engineering and Sciences (Fapesp/Cepid # 2013/08293-7-Brazil). We also acknowledge support from the sponsors of the Wave Inversion Technology (WIT) Consortium and Petrobras.

## References

Blumofe, R. D., and C. E. Leiserson, 1999, Scheduling multithreaded computations by work stealing: J. ACM, **46**, 720–748.

Dagum, L., and R. Menon, 1998, Openmp: An industry-standard api for shared-memory programming: IEEE Comput. Sci. Eng., **5**, 46–55.

de Kruijf, M., and K. Sankaralingam, 2011, Idempotent processor architecture: Presented at the Proceedings of the International Symposium on Microarchitecture (MICRO'11).

de Kruijf, M. A., K. Sankaralingam, and S. Jha, 2012, Static analysis and compiler design for idempotent processing: Presented at the Proceedings of the Conference on Programming Language Design and Implementation (PLDI'12).

Dean, J., and S. Ghemawat, 2008, Mapreduce: Simplified data processing on large clusters: Commun. ACM, **51**, 107–113.

Egwutuoha, I., D. Levy, B. Selic, and S. Chen, 2013, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems: The Journal of Supercomputing, **65**, 1302–1326.

Forum, M. P., 1994, Mpi: A message-passing interface standard: Technical report, Knoxville, TN, USA.

Jäger, R., J. Mann, G. Höcht, and P. Hubral, 2001, Common-reflection-surface stack: Image and attributes: Geophysics, **66**, 97–109.

Nickolls, J., I. Buck, M. Garland, and K. Skadron, 2008, Scalable parallel programming with cuda: Queue, **6**, 40–53.

Reinders, J., 2007, Intel threading building blocks, first ed.: O'Reilly & Associates, Inc.

Schlichting, R. D., and F. B. Schneider, 1983, Fail-stop processors: An approach to designing fault-tolerant computing systems: ACM Transactions on Computer Systems, **1**, 222–238.

Stone, J. E., D. Gohara, and G. Shi, 2010, Opencl: A parallel programming standard for heterogeneous computing systems: IEEE Des. Test, **12**, 66–73.