CUIRRE: An Open-source Library for Load Balancing and Characterizing Irregular Applications on GPUs

Tao Zhang[a,b,*], Wei Shu[a,b], Min-You Wu[a]

[a] *Department of Computer Science & Engineering, Shanghai Jiao Tong University, Shanghai 200240, China*
[b] *Department of Electrical & Computer Engineering, The University of New Mexico, Albuquerque 87131-0001, New Mexico, USA*

Abstract

While Graphics Processing Units (GPUs) show high performance for problems with regular structures, they do not perform well for irregular tasks due to the mismatches between irregular problem structures and SIMD-like GPU architectures. In this paper, we introduce a new library, CUIRRE, for improving performance of irregular applications on GPUs. CUIRRE reduces the load imbalance of GPU threads resulting from irregular loop structures. In addition, CUIRRE can characterize irregular applications for their irregularity, thread granularity and GPU utilization. We employ this library to characterize and optimize both synthetic and real-world applications. The experimental results show that a 1.63× on average and up to 2.76× performance improvement can be achieved with the centralized task pool approach in the library at a 4.57% average overhead with static loading ratios. To avoid the cost of exhaustive searches of loading ratios, an adaptive loading ratio method is proposed to derive appropriate loading ratios for different inputs automatically at runtime. Our task pool approach outperforms other load balancing schemes such as the task stealing method and the persistent threads method. The CUIRRE library can easily be applied on many other irregular problems.

*Corresponding author: Tao Zhang, 3-121 SEIEE Building, Shanghai Jiao Tong University, 800 Dong Chuan Road, Min Hang District,Shanghai 200240, China; Phone:+8613918520935

Email addresses: tao.zhang@sjtu.edu.cn (Tao Zhang), shu@ece.unm.edu (Wei Shu), mwu@sjtu.edu.cn (Min-You Wu)

Keywords:
Load Balancing, Characterizing, Library, Irregular, GPU

1. Introduction

In recent years, Graphics Processing Units (GPUs) have been used in various throughput-oriented computing applications, achieving remarkable speedups over their CPU counterparts. Examples include molecular dynamics, astrophysics simulation, life sciences, MRI reconstruction and so on. Among 1298 applications listed as August of 2013 at the NVIDIA CUDA showcase website [1], 137 reported speedups of at least 100×. However, some irregular applications, like 3D-lbm, Gafort [2] and N-queens solver [3] are not well-suited for being executed on GPUs, resulting in poor speedups. These applications generally exhibit a substantial amount of divergence. GPUs suffer divergence as they are inherently SIMDs that require all scalar pipelines execute the same instructions together. In particular, the architecture of the Streaming Processor (SM) in GPUs is Single-Instruction Multiple-Thread(SIMT). Besides divergence, load imbalance [3] is another major factor causing problems on GPUs. Due to the irregular and dynamic nature, each task (mapped onto a GPU thread) of an irregular application requires a different amount of computation, incurring load imbalance and hence performance degradation. Such imbalance cannot be eliminated by using a simple algorithmic redesign.

We optimized the irregular N-queens solvers on GPUs [3]. In fact we modified the application code manually to reduce the load imbalance. However, this process has been error-prone and tedious to repeat for each irregular application. It motivated us to design and implement this open-source CUIRRE library with convenient APIs for load balancing irregular applications. In addition, the library can characterize applications for their irregularity, thread granularity and GPU utilization. To further improve performance and applicability, we proposed an adaptive loading ratio method and a lightweight mode for the task pool approach. We characterized and optimized four irregular applications including N-queens solver, potential distribution and ray-tracing, which also demonstrated the effectiveness and conveniences of the library.

Overall, the contribution of this paper is as follows. First, we introduced the open-source CUIRRE library for characterizing and runtime load balancing irregular applications on GPUs. The library presents a centralized task

pool approach and a distributed task approach which outperformed other thread level load balancing schemes. Second, we proposed an adaptive loading ratio method and a lightweight mode for the task pool approach, which extended our previous work [3] and further improved applications' performance. Third, we characterized and optimized four irregular applications: Coulombic Potential(CP) [4], N-queens solver [4], potential distribution [5], and ray-tracing [1]. The source code of the CUIRRE library and sample applications can be downloaded from: http://wirelesslab.sjtu.edu.cn/download.html

2. Related Work

Recently, research has started to focus on performance optimization of irregular applications on GPUs [6]. It is often challenge for an irregular application to fully utilize the computing power of GPUs. Branch divergence and load imbalance were identified as two major issues to be addressed [3][7].

Work	**Method**	**Data structure**	**Handling new tasks**	**Fetch gran.**	**Lock**
Our Work [3]	centralized task pool	Single queue	no	thread	no
	distributed task pool	multiple queues	no	thread	no
Cederman et al. [8]	static task list	two arrays	yes	block	no
	blocking task queue	single queue	yes	block	yes
	lock-free task queue	single queue	yes	block	no
Cederman et al. [9]	task stealing	multiple queues	yes	block	no
Aila and Laine [10]	persistent threads	multiple queues	no	warp	no
Tzeng et al. [11]	task donation	multiple queues	yes	warp	no
Chen et al. [12]	task queue	two queues	yes	block	no

Table 1: Comparison of load balancing schemes on GPUs

Load balancing on GPUs has become a recent topic due to the advent of more elegant synchronization hardware components such as atomics, and the need to explore the power of GPUs on more complex graphics rendering or scientific computation. Load imbalance occurs on GPUs because the workload granularity of threads or warps are different, and/or tasks are newly generated during execution. Table 1 summarizes the most common load balancing schemes on GPUs. In 2008, Cederman et al. [8] proposed the static task list, blocking task queue, lock-free task queue and task stealing methods to address the load imbalance issue on GPUs at block level. The imbalance they addressed comes from an uneven distribution of jobs and newly generated jobs among the hardware processors. In 2009, Aila and Laine [10]

proposed a persistent threads method to balance the load at warp level. In 2010, Tzeng et al.[11] evaluated the above load balancing schemes and proposed a new task donation method to balance the load of Reyes pipeline application at warp level. However, the load imbalance of some irregular applications happens at thread level because of irregular loop structure. All these previous load balancing schemes work at either block [8] [9] [12] or warp [10] [11] level, and are therefore ineffective for the load imbalance at thread level. In 2011, we proposed a monolithic task pool approach and a distributed task pool approach [3] for load balancing at thread level. These approaches were the preliminary version of the task pool approaches developed in the CUIRRE library. Besides the work to balance the load within a single GPU, some other work has been conducted for load balancing among multiple GPUs. For example, Chen et al. [12] proposed a task queue approach to distribute tasks evenly among several GPUs.

There is some existing work that focuses on optimization of specific irregular applications on GPUs. Jia et al.[13] accelerated the Viola-Jones face detection algorithm using the persistent threads method and the Uberkernel method. Beck et al. [14] proposed a parallel interval Newton method on CUDA and adopted the static task list method for load balancing their application. In [3], we improved the performance of N-queens solvers by optimizing the memory access, eliminating the branch divergence and load imbalance. Frey et al. [7] reduced the branch divergence of fractals in the ray-casting isosurfaces application by a SIMT micro-scheduling approach. In this paper, we optimized several irregular applications including N-queens solver, potential distribution and ray-tracing.

There exist a few efforts on modeling or characterizing irregular workload. Wu et al. [15] studied several benchmarks to identify the sources of the control-flow irregularity. Rehman[16] studied the memory access and communication patterns of irregular applications, and proposed a model to predict their performance. Burtscher et al.[17] investigated the control-flow irregularity and memory-access irregularity using performance counters and summarized how irregular GPU kernels differ from regular kernels. The CUIRRE library can characterize irregular applications on their irregularity, resource utilization as well as the thread granularity.

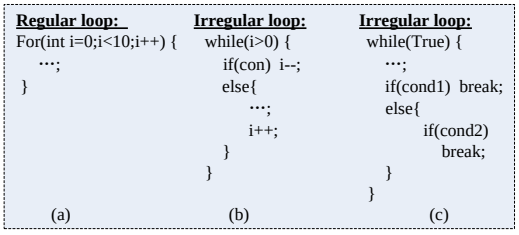


Figure 1: Regular and irregular loop structures

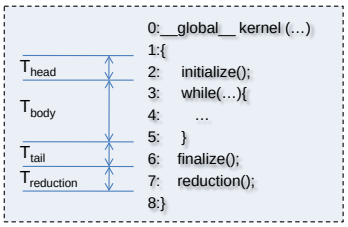


Figure 2: The structure of a kernel code

3. Irregular Loops and Thread-level Load Imbalance

The loop structures in a program fall into two categories: regular and irregular, as shown in Fig.1. A regular loop has a definite number of iterations while the number of iterations of an irregular loop depends on certain conditions. Assuming that the threads of a GPU warp are executing a loop structure simultaneously. Fig.1(a) is a regular loop in which all threads iterate 10 times. Figs.1(b) and (c) are irregular loops. In Fig.1(b), the termination of the loop depends on variable i whose value changes within the loop. Similarly, in Fig.1(c) threads within a warp could break out of the loop at different places. As a result, threads executing Fig.1(b) or Fig.1(c) take different number of iterations to finish the loop.

In SIMD-like GPUs, threads are grouped into warps and executed in a lock-step. A Streaming Processor (SM) achieves the full utilization when all threads within a warp exhibit the same behavior. However, threads behave differently when a warp is executing an irregular loop. Due to the post-dominator based re-convergence mechanism in GPUs [18], all threads within a warp need to re-converge after a loop. Consequently, after an irregular loop, threads completed earlier need to stall and wait for those later threads, resulting in a poor resource utilization. Irregular loops are often found in applications such as set covering problems, graph search, random walk, Monte-Carlo simulation or Ray-tracing. The task pool approach proposed here can significantly improve their performance.

3.1. Characterizing irregular applications

3.1.1. Structure of GPU kernels

We assume a code structure of irregular GPU kernels as shown in Fig. 2. In general, a kernel can be split into *head, body, tail* and an optional *reduction* part. The *body* part contains an irregular loop. The *head* usually

does some preparation such as variable initialization based on thread IDs. The *tail* part is often in charge of storing the results into the global memory. Finally, some kernels have a *reduction* part that performs a reduction across all threads within each block. Between any two adjacent parts, there could exist a barrier synchronization if needed. This code structure applies to all applications evaluated in this paper.

3.1.2. Irregularity

As shown in Fig. 2, the *body* part in a GPU kernel performs the major computation, which is to execute an irregular loop. Threads within each warp take different cycles to execute this loop. We define the *irregularity* of this loop as:

$$I_{irre} = \frac{m \times n}{\sum_{j=1}^{m} \sum_{i=1}^{n} T_{body}(i,j) / max(T_{body}(i,j))} \tag{1}$$

where n denotes the number of threads in each warp (normally 32 for Nvidia GPUs), m denotes the number of warps on the GPU, $T_{body}(i,j)$ denotes the number of clock cycles executed in the body part of thread i in warp j, $max(T_{body}(i,j)), i = 1...n.$ denotes the longest T_{body} in warp j. The I_{irre} characterizes the irregularity of a loop. A larger I_{irre} implies a more server load imbalance among threads. As a special case, a regular loop has its I_{irre} value equal to 1.

3.1.3. GPU utilization

The I_{irre} only considers the *body* part of a GPU kernel, therefore we define a U_{util} to model the GPU utilization of an entire kernel. As discussed in Sec. 3, based on the GPU's re-convergence mechanism, the execution time of a warp is calculated as:

$$T_{warp}(j) = T_{head}(j) + max(T_{body}(i,j)) + T_{tail}(j) + T_{reduction}(j) \tag{2}$$

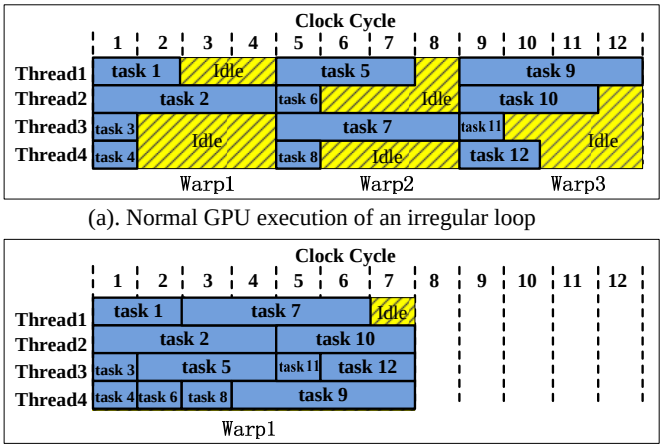
Where $T_{head}(j), T_{tail}(j), T_{reduction}(j)$ are the execution time of *head*, *tail*, and *reduction* parts of warp j, respectively. $max(T_{body}(i,j)), i = 1...n.$ denotes the longest T_{body} in warp j.

Correspondingly, the GPU utilization of an irregular kernel is:

$$U_{util} = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} T_{body}(i,j) + \sum_{j=1}^{m} (T_{head}(j) + T_{tail}(j) + T_{reduction}(j)) \times n)}{\sum_{j=1}^{m} T_{warp}(j) \times n} \tag{3}$$

In equation 3, the numerator denotes the cycles on computation while the denominator represents the cycles on computation and waiting. U_{util} considers not only the irregularity of the *body* part, but also the relative weight of the *body* part to the *head*, *tail*, and *reduction* parts in the kernel. In general, the smaller the U_{util} value, the larger the potential improvement using the CUIRRE library. The best possible value of U_{util} is 1.

4. Task Pool Approach for Thread-Level Load Balancing



(a). Normal GPU execution of an irregular loop

(b). GPU execution with the task pool approach

Figure 3: Illustration of the task pool approach

In [3], we proposed a preliminary task pool approach as illustrated in Fig. 3. For the sake of simplicity, we assume a warp size of 4 with 12 tasks executing the same irregular loop. Each thread handles exactly one task. As shown in Fig. 3(a), there are many waiting cycles in warps 1, 2 and 3 due to the unequal thread length and the post-dominator based re-convergence mechanism of GPUs. With the task pool approach, each thread is allowed to execute multiple tasks to balance the overall load within each warp. A thread can fetch a new task from the task pool upon completing its current task, as long as the pool is nonempty. As illustrated in Fig. 3(b), each thread processes multiple tasks and there is only one waiting cycle from thread 1. There are two schemes of the task pool approach: the centralized one and the distributed one.

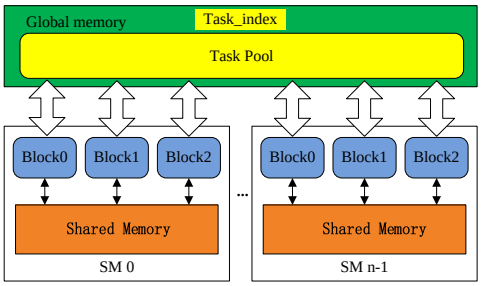


Figure 4: The centralized task pool approach

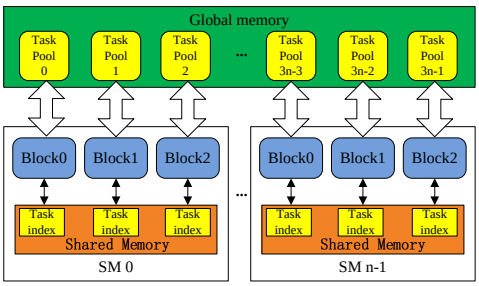


Figure 5: The distributed task pool approach

4.1. Centralized task pool approach

In the centralized task pool approach, there is a global task pool with a shared task index. The task pool and the task index are both stored in the GPU global memory, as shown in Fig.4. Threads access the task pool through an atomic operation on the task index, which maintains the index of the next available task in the task pool. Threads can fetch new tasks upon finishing their current tasks. Finally, the application ends when there are no more tasks in the task pool.

4.2. Distributed task pool approach

The distributed task pool approach is shown in Fig. 5. Each thread block has a task pool and a task index. The task pools are still located in the global memory while the task indices are now stored in the shared memory. There are NR tasks in each task pool except in the last one, where N is the block size (number of threads in each block) and R is the *loading ratio* (average number of tasks executed by each thread).

4.3. Performance analysis of two task pool approaches

Since all threads share a task pool, the centralized task pool approach can reach a better load balance than a distributed scheme. Besides, the centralized task pool approach introduces less overhead on boundary checking when fetching a new task – it only ensures that the new task index is smaller than the total task count while the distributed scheme needs to perform additional checking to make sure that the new task index is also within the boundary of the block's task pool. As a result, there are more branches if using the distributed task pool approach, as demonstrated by the NVIDIA

Visual Profiler. In addition, the centralized task pool approach exhibits a better locality in memory accesses. On the other hand, the distributed task pool approach takes less cycles to access task indices since they are stored in the shared memory instead of the global memory. In order to determine which scheme is the best, we need to evaluate them with experiments.

4.4. Adaptive loading ratio method

In the task pool approach, there is a *loading ratio* R which represents the average number of tasks executed by each thread. In this section, we first analyze the impact of loading ratios on performance qualitatively, then propose an adaptive loading ratio method based on the analysis.

4.4.1. Impact of loading ratios

The loading ratio is the only parameter in the task pool approach which has a significant impact on performance. In the CUDA programming and computation model, there are multiple thread blocks running on a GPU simultaneously. A thread block contains one or more warps, each consisting of multiple threads working in a SIMD manner. Let N denote the total number of tasks and R denote the task loading ratio. Let $warp\_size$ denote the number of threads in a warp (normally 32 in Nvidia GPUs) and $block\_size$ denote the number of threads in a block. Let $\#warps$ and $\#blocks$ denote the total number of warps and thread blocks executing this GPU kernel, respectively. Then we have:

$$\#warps = \frac{N}{R * warp\_size} \tag{4}$$

$$\#blocks = \frac{N}{R * block\_size} \tag{5}$$

A modern GPU has multiple SMs and each SM processes multiple warps simultaneously. The processing throughput of the GPU can be defined as P warps per second. For a specific application and input running on a GPU, there exists a highest achievable P called P_{max}. To reach P_{max}, at least a threshold number of warps or thread blocks must be present concurrently on the GPU such that the GPU has enough parallelism to hide the pipeline and memory access latency [19]. We call this minimum number of required thread blocks as **block threshold**. For example, the *block threshold* for our Nvidia

GTX480 GPU is around 90 (6 blocks per SM $\times$ 15 SMs). The processing throughput P is given by:

$$
\begin{aligned}
P &= \begin{cases} P_{max} & \text{if \#blocks} \geq \text{block threshold}, \\ P_{max} * \frac{\#blocks}{block \quad threshold} & \text{otherwise.} \end{cases} \\
&= \begin{cases} P_{max} & \text{if } \frac{N}{R*block\_size} \geq \text{block threshold}, \\ P_{max} * \frac{N/(R*block\_size)}{block \quad threshold} & \text{otherwise.} \end{cases}
\end{aligned} \tag{6}
$$

Equation 6 is a simplified definition since in reality, P often changes during different phase of the computation rather than stays at P_{max} constantly. However, this definition does not affect the correctness of a qualitative analysis of the loading ratio's impact on performance.

With the task pool approach, when a thread is fetching a new task, all other 31 threads in its warp would stall due to the branch divergence. Therefore, the running cycles of warp j (denoted as $T_{warp}(j)$) can be categorized into fetching cycles $T_{fetch}(j)$ and non-fetching cycles $T_{exec}(j)$. Fetching cycles $T_{fetch}(j)$ are the cycles that any thread within warp j is fetching a new task, while non-fetching cycles $T_{exec}(j)$ are the execution cycles. That is:

$$T_{warp}(j) = T_{exec}(j) + T_{fetch}(j) \tag{7}$$

The sum of the run time of all warps T_{warp_sum} is:

$$
\begin{aligned}
T_{warp\_sum} &= \sum_{j=1}^{\#warps} T_{warp}(j) \\
&= \sum_{j=1}^{\#warps} T_{exec}(j) + \sum_{j=1}^{\#warps} T_{fetch}(j) \\
&= \sum_{j=1}^{\frac{N}{R*warp\_size}} T_{exec}(j) + \sum_{j=1}^{\frac{N}{R*warp\_size}} T_{fetch}(j)
\end{aligned} \tag{8}
$$

In the above equation, there are two parts at the right of the equal sign. The first part corresponds to the sum of the execution time of all warps, and the second part is the sum of the task fetching time of all warps, which is the overhead. Since there are N/R threads each having an initial task, there will

be $N - \frac{N}{R} = N * \frac{R-1}{R}$ tasks to be fetched from the task pool. Assuming each fetch of a new task takes t_{fetch} time, we can substitute the second part of the equation 8 by $\frac{R-1}{R} * N * t_{fetch}$ and get:

$$T_{warp\_sum} = \sum_{j=1}^{\frac{N}{R*warp\_size}} T_{exec}(j) + \frac{R-1}{R} * N * t_{fetch} \tag{9}$$

Finally, we can estimate the run time T of the kernel using:

$$\begin{aligned} T &= \frac{T_{warp\_sum}}{P} \\ &= \frac{\sum_{j=1}^{\frac{N}{R*warp\_size}} T_{exec}(j) + \frac{R-1}{R} * N * t_{fetch}}{P} \end{aligned} \tag{10}$$

There are two parts in the numerator of equation 10. The first part is the time for warps to process N tasks, which decreases with increased loading ratios. With task pool approaches, each GPU thread processes multiple tasks. As the increasing of the loading ratio, there are less waiting cycles (as shown in Fig. 3) and application performance becomes better depending on the resource utilization/irregularity of the application. In general, applications with lower GPU utilization (or larger irregularity) can achieve more significant performance improvement. We call this performance improvement the **benefit** of task pool approaches. The second part is the time to fetch new tasks (the **overhead**), which increases with increased loading ratios. Thus, application performance depends on which part dominates: the benefit or the overhead. Finally, according to equation 6, the denominator of equation 10 is a constant when the loading ratio is small. However, when the loading ratio keeps increasing and ultimately there will be insufficient blocks on the GPU, P will decrease and application performance drops rapidly.

In summary, the impact of loading ratios on performance is illustrated in Fig. 6. In phase 1, application performance keeps increasing with the growing of the loading ratio. In phase 2, application performance drops off rapidly since the loading ratio becomes too large and the number of blocks is less than the block threshold. Therefore, applications should always adopt proper loading ratios that keep enough thread blocks on GPUs. We summarize the relationship between application's resource utilization, parallelism (number of tasks) and performance improvement in Table 2. Generally speaking, more significant performance improvement can be achieved for applications with high parallelism and low GPU utilization.

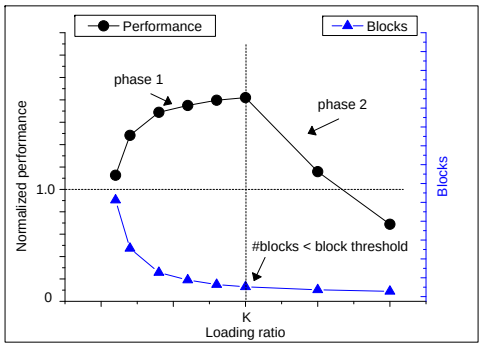


Figure 6: The impact of loading ratios on performance

Parallelism	Utilization	Performance improvement
high	low	high
high	high	low
low	low	medium
low	high	low

Table 2: The relationship between the parallelism, resource utilization and performance improvement

4.4.2. Adaptive loading ratios

Selection of an appropriate loading ratio is crucial in order to secure the benefit of the task pool approach. Exhaustive search of a best loading ratio is expensive and impractical. In addition, applications with different characteristics and various input require different loading ratios to achieve the optimal performance.

We propose an adaptive loading ratio method to derive proper loading ratios automatically at runtime. It adopts the loading ratio that keeps just enough blocks on a GPU ($\#blocks = block\ threshold$). The loading ratio is calculated at runtime based on equation 5 and the capability of the GPU including the number of SMs and the hardware limit for simultaneously active blocks in each SM.

There are two major benefits of the adaptive loading ratio method:

- It determines a proper loading ratio automatically at runtime, saving efforts and cost for a exhaustive search of the optimal loading ratio.
- It adaptively derives different loading ratios for different inputs or problem sizes of an application, achieving better performance than static loading ratios.

We will evaluate the performance of the loading ratio method in Sec. 8.3.

5. CUIRRE Library

The CUIRRE library is designed to facilitate the characterizing and optimization work of irregular applications (CUIRRE stands for CUDA for

IRREgular applications). It implemented a task management system for applications and provided a set of simple, convenient APIs. The library consists of three modules as summarized in Table 3.

Module	Function	Platform	Overhead	Dependency
Load balancing	load balancing	CPU & GPU	Yes	None
Diagnostics	generating thread trace	CPU & GPU	Yes	None
Characterizing	analyzing thread trace	CPU	No	Diagnostics module

Table 3: The summary of modules

The load balancing module works independently to provide the load balancing functionality. The diagnostics module generates the application thread trace, which is the input for the characterizing module to analyze applications' characteristics. An application can either apply the load balancing module to optimize performance, or apply the diagnostics module to generate thread trace for analysis. The load balancing module and the diagnostics module are never applied at the same time.

5.1. Load balancing module

The APIs of the load balancing module are listed in Table 4. These APIs can be categorized into three types: APIs for the centralized task pool approach, APIs for the distributed task pool approach and commonly shared APIs. Besides, both the centralized and the distributed task pool scheme can work in two modes: full and lightweight. In the full mode, the library allocates and de-allocates the task pool, and manages both the task data and the task index. User applications submit the task data to the library by calling the *cuirreSubmitTasks()* API. In the lightweight mode, however, the library only manages the task index. The task data is managed by user applications themselves. APIs for the lightweight mode have an "LW" suffix. Selection of the full mode or the lightweight mode APIs depends on applications' needs. For example, among the applications we evaluated, N-queens solver employs the full mode APIs while CP, potential distribution, and ray-tracing all adopt the lightweight mode APIs. The lightweight mode of the task pool approach is extended from our previous work in [3].

5.1.1. Overhead analysis

As shown in equation 10, the fetching cycles are the overhead compared to the real execution cycles. We define the overhead of the load balancing

APIs shared by all task pool approaches
*CUIRREBoolean CUIRRE_API cuirreConfig(CUIRREConfig_T *config);* Config the cuirregular library parameters including load balancing scheme selection, number of tasks and size of tasks.
dim3 CUIRRE_API cuirreGetBlockDim(); Return the dimension of a block.
dim3 CUIRRE_API cuirreGetGridDim(); Return the dimension of the grid.
*CUIRREBoolean CUIRRE_API cuirreAllocateTaskPool(CUIRRE_TASK** taskPool);* Allocate memory for the task pool(s).
void CUIRRE_API cuirreDeallocateTaskPool(CUIRRE_TASK taskPool);* Release the memory allocated for the task pool(s).
CUIRREBoolean CUIRRE_API cuirreSubmitTasks(CUIRRE_TASK taskPool, void* tasks);* Submit tasks to the library for managing.
APIs for the centralized task pool approach
void CUIRRE_API cuirreSetGlobalTaskIndex(size_t index); Initialize the global task index.
__device__ inline CUIRREBoolean CUIRRE_API cuirreFetchATaskCentralized(CUIRRE_TASK taskPool,CUIRRE_TASK* task, unsigned int* index = NULL);* Fetch a new task from the global task pool; full mode API.
__device__ inline CUIRREBoolean CUIRRE_API cuirreFetchATaskCentralizedLW(unsigned int index);* Fetch a new task from the global task pool; lightweight mode API.
APIs for the distributed task pool approach
void CUIRRE_API CUIRRE_DECLEAR_BLOCK_SEEK(); Declare the task index variable in shared memory for each thread block.
__device__ inline void CUIRRE_API cuirreSetBlockTaskIndex(); Initialize the task index of each thread block.
__device__ inline CUIRREBoolean CUIRRE_API cuirreFetchATaskDistributed(CUIRRE_TASK taskPool,CUIRRE_TASK* task,unsigned int* index);* Fetch a new task from the task pool; full mode API.
__device__ inline CUIRREBoolean CUIRRE_API cuirreFetchATaskDistributedLW(unsigned int index);* Fetch a new task from the task pool; lightweight mode API.

Table 4: Load balancing APIs

module as the percentage of fetch cycles in total running cycles of all warps, as following:

$$overhead = \frac{\frac{R-1}{R} * N * t_{fetch} * 100}{\sum_{j=1}^{\frac{N}{R*warp\_size}} T_{exec}(j) + \frac{R-1}{R} * N * t_{fetch}}\% \tag{11}$$

Where the symbols follow the same definition as in Sec. 4.4.1. This overhead cannot be obtained on a real GPU with common profiler tools. Therefore we measure the overhead with a cycle-accurate GPGPU simulator GPGPU-Sim [4]. GPGPU-Sim is widely used by researchers to publish papers [20] [21] [22] in top architecture conferences such as Micro, HPCA and ISCA. We modify the simulator to let it record the fetching cycles and the overall cycles for every warp. The results are shown in Sec.8.2.

5.2. Diagnostic module

Diagnostic APIs
CUIRRE_DIAG_DECLARE_VARIABLES() Define three variable "diag_data, diag_data_cuda, and diag_fp" for the diagnostic purpose.
CUIRRE_DIAG_CONFIG(nTasks) Configure diagnostics parameters.
CUIRRE_DIAG_ALLOCATE(size) Allocate memory for diag_data and diag_data_cuda, and open a file for saving the thread trace.
CUIRRE_DIAG_GET_DATA(size) Copy data from the GPU memory to the host memory and save into the thread trace file.
CUIRRE_DIAG_DEALLOC() Release the memory allocated for diag_data and diag_data_cuda, and close the thread trace file.
CUIRRE_DIAG_BEGIN(diagData, index, tBegin) Record the beginning time of a task.
CUIRRE_DIAG_END(diagData, index, tEnd) Record the finish time of a task.

Table 5: Diagnostic APIs

The APIs of the diagnostics module are listed in Table 5. The major functionality of this module is to output thread traces for analysis. Normally we apply this module on original GPU application implementations (using no load balancing schemes; one task per thread). The *CUIRRE_DIAG_BEGIN()* is called at the beginning of an irregular loop (outside the loop) in the GPU kernel and the *CUIRRE_DIAG_END()* is called at the end of the irregular loop (inside the loop). Then the elapse time between these two calls gives the computation time of this task.

5.2.1. Overhead analysis

The *CUIRRE_DIAG_BEGIN()* API calls an internal clock() function to get the current clock cycle. It incurs no branch divergences since all threads within each warp call this function at the same time. The *CUIRRE_DIAG_END()* API calls the internal clock() function again and calculates the elapsed clock cycles. It introduces branch divergences since threads complete the irregular loop and call this API at different time. Normally these two APIs will be called only once in the execution of an irregular loop on a GPU. Let T_{diag} and $T_{baseline}$ be the run time of the GPU code with and without enabling this diagnostics module respectively, then we define the overhead of the diagnostics module as $((T_{diag} - T_{baseline})/T_{baseline})$. The results are shown in Table 12 in Sec. 8.1.3.

Characterizing APIs
void CUIRRE_API cuirreCalcIrregularityFromFile(char filename, unsigned int &nTasks, float &irregularity);* make statistics and calculate the irregularity.
void CUIRRE_API cuirreCalcUtilizationFromFile(char taskfile, char* configfile,float &utilization, unsigned int &thread_average);* make statistics and calculate the utilization.

Table 6: Characterizing APIs

5.3. Characterizing module

The characterizing APIs are listed in Table 6. The *cuirreCalcIrregularityFromFile()* reads a file containing the thread trace generated by the diagnostics module. It then reports the number of tasks, the minimum, maximum and average task length in clock cycles and the irregularity.

cuirreCalcUtilizationFromFile() reads the same file and outputs the number of threads and warps, the minimum, maximum and average thread length in clock cycles, and the utilization of GPU resources. The thread length includes the computation cycles and waiting cycles introduced in Sec. 3.

5.3.1. Overhead analysis

The characterizing module introduces no performance overhead to the GPU kernel since the module runs only on CPUs.

5.4. Applications of the CUIRRE library

The CUIRRE library is applicable to general irregular applications suffering thread-level load imbalance regardless of the hardware platform (PCs or mobile devices). For example, most mobile phone chips such as Nvidia Tegra have integrated GPUs to accelerate graphics rendering as well as general purpose computing. There are irregular loop structures in games using ray-tracing or applications using graph algorithms such as BFS, so the library can be applied to optimize or characterize these applications.

5.4.1. Load balancing module

The load balancing module is especially useful for coarse-grained threads with substantial irregularity. In some dynamic and/or irregular applications, the program size/space can be quite large and unpredictable. Therefore they often employ course-grained threads to search in the solution space. The load balancing module would be useful for such applications.

Fig. 7 shows the pseudo code to apply the full mode APIs or the lightweight mode APIs of the centralized task pool approach. In general, applications

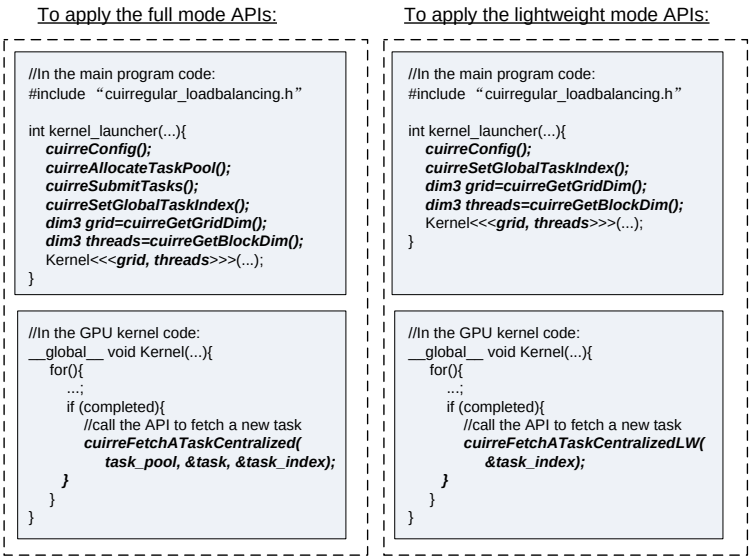


Figure 7: The pseudo code to apply the centralized task pool APIs

need to include a header file containing API declarations of the CUIRRE library, and call the APIs to config and fetch tasks. Finally applications can be compiled and linked to the library for execution. There are documents and two sample applications (CP and N-queens solver) in the CUIRRE library downloading to illustrate the usage of the library.

5.4.2. Diagnostics and characterizing module

The diagnostics module and the characterizing module provide useful information and have various applications – bottleneck projection, irregularity comparison, warp scheduling and irregularity study, to name a few.

- *bottleneck projection:* characterizing several problem sizes to predict applications' behavior and bottleneck for larger problem sizes. For example, by analyzing the evaluation results of N-queens solver in Table 9 and Fig.11, we can predict that for larger problem size (eg. 21 and higher), the parallelism and the irregularity of the N-queen problem will increase while the thread granularity and GPU utilization will decrease. The bottleneck of N-queen solver is the irregularity but not the parallelism. For potential distribution under more particles, the parallelism and thread granularity will grow up while the irregularity will drop off.

- *irregularity comparison:* to judge which application is more irregular among applications, or an application is more irregular under which input.

- *warp scheduling:* the data such as the average, minimal and maximal thread granularity can be used to aid the design and execution of warp scheduling algorithms on GPUs. For example, Lee et al.[23] proposed a criticality-guided priority scheduling algorithm for scheduling irregular applications and achieved 20% performance improvement on a BFS application. The basic idea is to give warps having longer threads more opportunity to run. The data about the average, minimal and maximal task granularity are exactly the criticality for that scheduling algorithm.

- *Irregularity studying:* the thread trace generated by the diagnostics module can be used to study the characteristics of irregular applications such as the distribution of their thread granularity.

6. Workloads

We characterized and optimized a synthetic application and three real world applications using the CUIRRE library. With the synthetic workload, we can adjust the thread granularity and the irregularity and observe the corresponding performance. We use a modified version of Coulombic Potential (CP) [24] [4] as the synthetic workload. For real world workloads, we optimized N-queens solver [4][3], potential distribution [5] and ray-tracing [1].

6.1. Synthetic workload

Coulombic Potential (CP) is part of a legacy version of the Parboil Benchmark suite [24] developed by the IMPACT research group at UIUC. The code we used is distributed with GPGPU-SIM [4] as one of the default benchmarks. CP is useful in the field of molecular dynamics. Loops are manually unrolled to reduce the loop overheads and the point charge data is stored in constant memory to take advantage of caching. The CP has been heavily optimized. We test 4000 atoms on a grid size of 1024×1024.

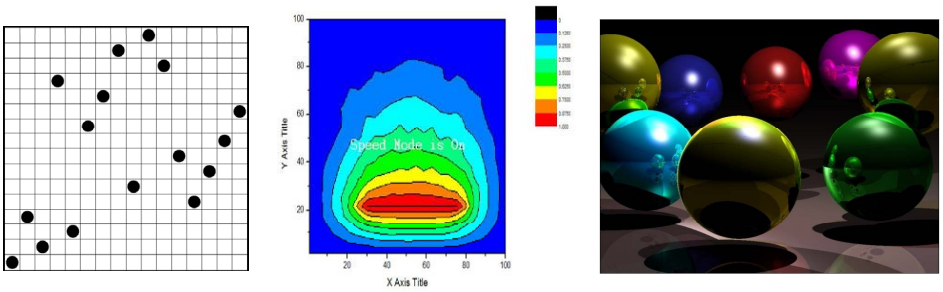
We modified the code of CP to adjust the thread granularity and the irregularity of the *body* part. The application accepts a command line parameter: the granularity exponent E. At initialization, the application reads

a file containing pre-generated random numbers between 0.0 and 1.0 into a $Coef$ array. Then it performs the computation:

$$Coef[i] = ceil(pow(Coef[i] \times 10.0, E))$$

Finally, $Coef[i]$ will be used as the number of iterations of the loop in the body part of GPU thread i. As a result, the irregularity of CP increases rapidly with E.

6.2. Real world workloads



(a) N-queens solver (b) Potential distribution (c) Ray-tracing

Figure 8: Three real world applications

6.2.1. N-queens solver

N-queens solver searches all distinct solutions to place N queens on an $N \times N$ chess board such that there is no two queens at a same column, row or diagonal. Fig. 8 (a) shows a solution for 16-queens problem. The total solution cost for the N-queens problem increases exponentially with N [25]. As a result, it is classified in the NP (Non-Deterministic Polynomial) complexity class [26]. Some research projects use massively-parallel FPGA-based devices or supercomputers to solve the N-queen problem for large N [27].

The implementation we used is distributed with GPGPU-SIM [4] as one of the default benchmarks. In the implementation, the CPU always solves the first $(N-11)$ rows while the GPU solves the rest 11 rows. The size of the shared memory limits the number of rows a GPU can process. Each GPU thread processes a partial solution generated on the CPU and performs a exhaustive search to find out all valid placement on the rest 11 rows. The

search process is an irregular loop. We executed 16-queen to 20-queen in the experiments. Since the number of tasks grows rapidly with N (see Table 9 in Sec.8.1), the N-queen solver generates tasks on the CPU and sends them to the GPU in batches of 1152000 tasks since otherwise there will be insufficient memory to hold all tasks of a large-N N-queens problem and the number of tasks would exceed the limit of threads on the GPU. For example, the total tasks of 17-queens problem and 18-queens problem will be sent to the GPU in two and 13 batches, respectively. In addition, this batch processing enables the overlaping of computation and data transferring between the CPU and GPU.

6.2.2. Potential distribution

The potential distribution application [5] calculates the distribution of the potential in a field using the Monte-Carlo simulation method, as shown in Fig. 8 (b). The field is a cross section of a rectangular waveguide. Each GPU thread simulates a particle in the field that walks step by step following a probabilistic model until satisfying the termination condition. The Monte-Carlo simulation method is widely used in physical and mathematical problems when it is impossible to obtain a closed-form expression or infeasible to apply a deterministic algorithm [28].

The Monte-Carlo simulation process in this application is an irregular loop where threads take different cycles. Actually most Monte-Carlo simulations contain irregular loops and therefore the task pool approach is applicable. The simulation process is repeated N times ($N = 80$ in the experiment) to improve the accuracy of the results. The input of this application is the number of particles, and we vary this number to 19600, 22500, 25600, 28900, 32400 in the experiments. In order to proceed to a fair performance comparison of different runs, we use pseudo-random numbers instead of real random numbers.

6.2.3. Ray-tracing

Ray-tracing [1] is a method of rendering graphics with near photo-realism, as shown in Fig.8 (c). In this implementation, each 24 pixels rendered corresponds to a scalar thread on the GPU. Up to five levels of reflections and shadows are taken into account, so the thread behavior depends on what object the ray hits (if it hits anything at all), making the kernel susceptible to branch divergence and load imbalance. We tested rendering five scenes of different combinations of dimension (1024×960 to 1536×1152), num-

Configuration items	Value
SMs/CUDA processors	15/480
Warp size	32
Capacity / Core	MAX. 1536 Threads, 8 CTAs
Core / Memory clock	700 MHz / 924 MHz
Interconnection network	1.4 GHz, 32 bytes wide, crossbar
Registers / Core	32768
Shared memory / Core	48KB
Constant cache / Core	8KB, 2-way, 64B line
Texture cache / Core	4KB, 24-way, 128B line
L1 data cache / Core	32KB, 4-way, 128B line
L1 I-cache / Core	4KB, 4-way, 128B line
L2 cache	64KB, 16-way, 128B line
Warp scheduler	Greedy then Oldest(GTO)
DRAM model	FR-FCFS memory scheduler, 6 memory modules

Table 7: Simulator architectural configuration

ber of objects (9 - 18) and number of lights (4 - 8). In addition, the objects have different material, color reflection attribute and diffusion attribute. The rendering process is an irregular loop and threads take different cycles.

7. Experiment Methodology

In this section, we describe the experiment methodology including the hardware and software platform, the testing process, other load balancing schemes, comparison metrics and the method for overhead measurement.

7.1. Hardware and software platform

We use an NVIDIA GeForce GTX480 GPU card to evaluate application performance with the CUIRRE library. NVIDIA GeForce GTX480 is one of the most powerful products in the GeForce series of Fermi architecture [29]. It consists of 480 CUDA processors, each having a fully pipelined integer Arithmetic Logic Unit (ALU) and a Floating Point Unit (FPU). These 480 processors are grouped into 15 Streaming Multiprocessors (SMs), each of which contains 32 processors. There are two instruction units in each SM, one for every 16 processors. In every clock cycle, each of the two instruction units selects an ready instruction and issues it to a group of 16 processors or 16 load/store units or four Special Function Units (SFUs). This is called the dual issue [29].

All experiments are executed on the NVIDIA GTX480 GPU except the overhead measuring of the load balancing module. We use GPGPU-Sim [4] 3.2.1 and the official configuration files to simulate a GTX480 architecture for the overhead measuring of the load balancing module. The architectural settings are shown in Table 7. For the consistency of the experiment results on the GTX480 GPU and on the simulator (to ensure the results are comparable), we use NVIDIA CUDA 4.2 [30] in all experiments since this is the highest CUDA version supported by the GPGPU-Sim.

7.2. Testing process

We repeat each experiment ten times and report the average results. To perform a fair comparison, all the implementation use a block size of 64. Each application uses a static loading ratio R for its centralized or distributed task pool implementation. In addition, this R is also used as the size of the thread task queue in the task steal implementation. Different application can adopt different R. In the persistent threads implementation of applications, we launch only enough threads to fill the GPU once, and let the thread 0 of each warp fetch new tasks for its warp threads.

7.3. Load balancing schemes for comparison

7.3.1. Task stealing method

The task stealing method on GPUs was proposed by Cederman et al.[8][9] to handle the load imbalance at block level. In their paper, each thread block is assigned an initial set of tasks and can steal new tasks from other blocks after completing the initial tasks. In this study, we need to address the load imbalance at thread level instead of block level. Therefore, we extend their method and let each thread have its own queue with an equal number of initial tasks. When the queue of a thread is empty, it tries to steal new tasks from the queue of other threads in a round-robin fashion, eg. thread i looking at thread $i+1$ followed by $i+2$ and so on. A thread can steal tasks only from threads within its block. Besides, since each block is executed on a SIMT processor, other threads within this block will stall when a thread is stealing tasks.

7.3.2. Persistent threads method

The persistent threads method was proposed by Aila and Laine [10] to handle the load imbalance at warp level. In GPUs, the execution time of a block is determined by the longest warp. A new block can only be launched

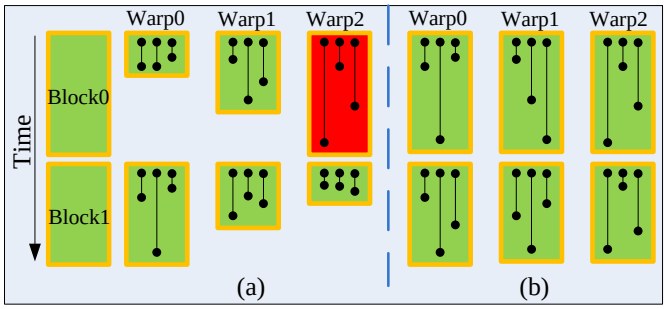


Figure 9: (a)an application having both warp-level and thread-level load imbalance. (b)an application having only thread-level load imbalance

after the completion of an old block. In Fig.9 (a), warp 0 and warp 1 have to wait for warp 2 to finish. Block 1 can not be launched until the completion of warp 2 in block 0, causing starvation issues in the GPU work distribution unit. The persistent threads method launches only enough threads to fill the GPU once. These long-running persistent threads can then fetch new tasks from a global pool using an atomic counter until the pool is empty. In their design, the fetching of new tasks happens at warp level. Thread 0 of a warp will fetch new tasks for all threads within this warp when all of them finish their prior tasks.

Irregular applications often have thread-level and warp-level load imbalance at the same time. In fact the different thread granularity is the root cause, as shown in Fig.9 (a). The persistent threads method can fix the low utilization due to warp-level load imbalance, but is ineffective in fixing the load imbalance at thread-level. On the contrary, our task pool approaches addresses the thread-level load imbalance directly, and thus also indirectly addresses the warp-level imbalance. Moreover, Fig. 9 (b) shows a special case where an irregular application has only thread-level load imbalance. The persistent threads method is ineffective for such case.

7.4. Comparison metric

We use the execution time of application kernels as the performance metric. Each application has an un-optimized implementation as the baseline, whose kernel execution time is $T_{baseline}$. In addition, each application has four optimized implementations with one of the following load balancing schemes: the centralized task pool approach, the distributed task pool approach, the task steal approach and the persistent threads approach. Then performance

of an optimized implementation is normalized to the unoptimized baseline implementation:

$$Normalized \quad performance = \frac{T_{baseline}}{T_{optimized}}$$

.

7.5. Overhead measurement

For the overhead of the load balancing module, we make the measurement on GPGPU-Sim [4]. We measure the total running cycles T_{run} and the cycles to fetch tasks T_{fetch} for each warp. Then we calculate the average T_{fetch}/T_{run} for all warps as the overhead. The results are shown in Sec.8.2.

For the overhead of the diagnostics module, we do the measurement on the GTX480 GPU with the baseline implementations (without using any loading schemes; one task per thread). First, we run the baseline implementation of an application with the diagnostics module disabled and record the run time as $T_{baseline}$. Then we enable the diagnostics module in the baseline application and record the run time as T_{diag}. Then the overhead is calculated as: $(T_{diag} - T_{baseline})/T_{baseline}$. The results are shown in Table 12 in Sec. 8.1.3.

8. Results and Analysis

8.1. Characterization of applications

This section presents the results on application irregularity, GPU utilization and thread granularity. The irregularity and GPU utilization is calculated by the characterizing module based on the thread trace generated by the diagnostics module running the baseline implementations.

8.1.1. Irregularity and GPU utilization

Fig. 10, Fig. 11, Fig. 12 and Fig. 13 show the irregularity and GPU utilization of the four applications under different inputs. CP and potential distribution have higher average irregularity of 2.× and lower average GPU utilization among all applications. N-queens solver and ray-tracing have relatively lower average irregularity of 1.× and higher average GPU utilization. Applications with lower GPU utilization such as CP have more room for optimization. Besides, we can observe different trends on applications. For example, under larger inputs, the irregularity of N-queens solver grows up while that of potential distribution drops off.

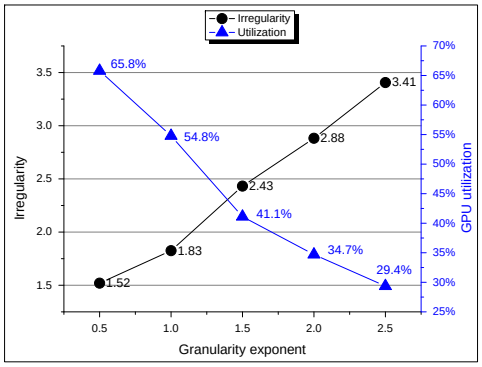


Figure 10: The irregularity and GPU utilization of CP

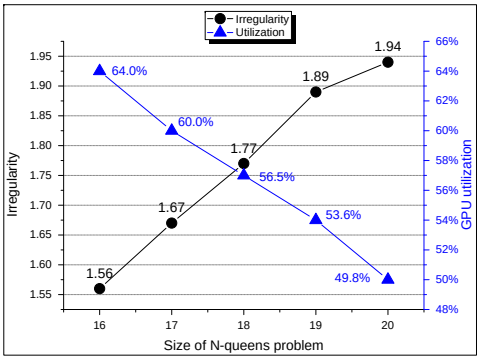


Figure 11: The irregularity and GPU utilization of N-queens solver

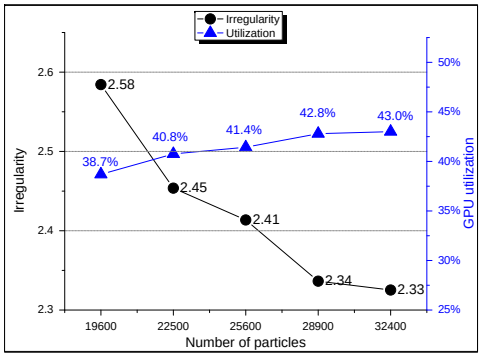


Figure 12: The irregularity and GPU utilization of potential distribution

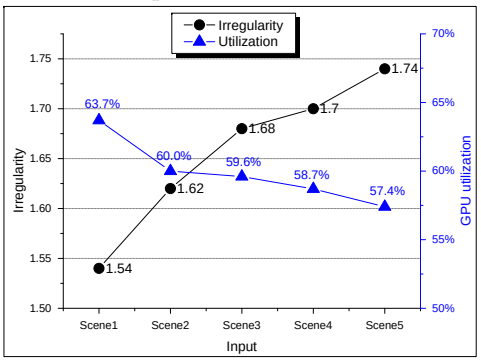


Figure 13: The irregularity and GPU utilization of ray-tracing

Granularity exponent	#Tasks	Average	Average'
0.5	1.31E+05	3.96E+07	6.04E+07
1.0	1.31E+05	8.48E+07	1.55E+08
1.5	1.31E+05	2.16E+08	5.25E+08
2.0	1.31E+05	5.79E+08	1.67E+09
2.5	1.31E+05	1.15E+09	3.93E+09

Table 8: The thread granularity of CP

Number of particles	#Tasks	Average	Average'
19600	1.93E+04	1.18E+09	3.02E+09
22500	2.21E+04	1.32E+09	3.20E+09
25600	2.52E+04	1.40E+09	3.35E+09
28900	2.88E+04	1.47E+09	3.45E+09
32400	3.23E+04	1.52E+09	3.51E+09

Table 10: The thread granularity of potential distribution

Size of N-queens	#Tasks	Average	Average'
16	1.42E+05	9.46E+06	1.49E+07
17	1.45E+06	6.53E+06	1.11E+07
18	1.48E+07	4.64E+06	8.46E+06
19	1.54E+08	2.33E+06	5.60E+06
20	1.62E+09	1.42E+06	3.23E+06

Table 9: The thread granularity of N-queens solver

Input	#Tasks	Average	Average'
Scene1	6.14E+04	8.61E+06	1.48E+07
Scene2	6.56E+04	5.90E+06	1.04E+07
Scene3	7.37E+04	4.26E+06	7.39E+06
Scene4	4.61E+04	5.53E+06	9.63E+06
Scene5	4.10E+04	4.53E+06	8.06E+06

Table 11: The thread granularity of ray-tracing

8.1.2. Parallelism and thread granularity

Table 8, Table 9, Table 10 and Table 11 show the parallelism (number of tasks) and the thread granularity of the four applications. The column *average* is the average clock cycles excluding the waiting cycles of GPU threads to execute the irregular loop. The column *average'* is the average clock cycles including the waiting cycles of GPU threads to execute the irregular loop. These two values represent the average thread granularity with and without considering the waiting cycles due to re-convergence. Overall, potential distribution has the largest average thread granularity. Besides, N-queens solver has much higher parallelism than the rest applications, so it can support larger loading ratio and still maintains enough thread blocks on the GPU. In addition, the parallelism of N-queens solver grows rapidly with the size of N-queens. On the contrary, the parallelism of CP keeps constant since its input affects only the thread granularity and irregularity.

8.1.3. Overhead of the diagnostics module

Application / Input	**1**	**2**	**3**	**4**	**5**
CP	0.26%	0.29%	0.20%	0.31%	0.69%
N-queens solver	0.61%	1.10%	1.01%	0.80%	0.95%
Potential distribution	3.43%	1.69%	2.58%	2.87%	3.12%
Ray-tracing	2.15%	2.45%	1.19%	1.13%	1.02%

Table 12: The overhead of the diagnostics module

The overhead of the diagnostics modules is shown in Table 12. Each application uses the same five inputs as in previous experiments. Potential distribution has the largest overhead of 2.73% on average and up to 3.43%. In contrast, CP has a very small overhead of 0.35% on average and up to 0.69%. Because of this overhead, the diagnostics module is disabled by default. We only turn it on when we need to generate the thread trace.

8.2. Performance comparison

In this section, we evaluation the performance of load balancing schemes. In performance comparison plots, we also draw the standard deviation error bars. The standard deviations of performance are quite small (between 1.00E-6 and 9.99E-3 in our experiments) since the execution time on GPUs is quite stable. Therefore, we also list the standard deviation of the centralized or distributed task pool scheme in tables.

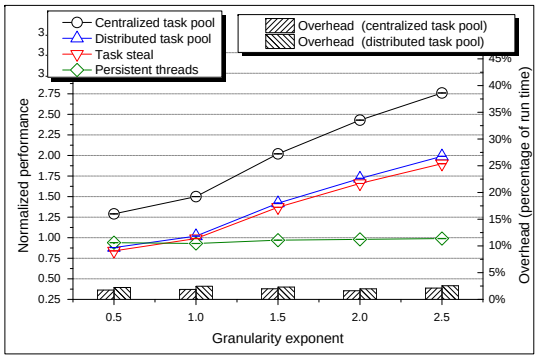


Figure 14: Performance comparison of CP (loading ratio = 16)

Scheme	Input	Perform.	SD
Centralized task pool	0.5	1.29	1.47E-3
	1.0	1.50	5.26E-4
	1.5	2.03	1.29E-3
	2.0	2.43	9.58E-4
	2.5	2.76	6.41E-4
Distributed task pool	0.5	0.88	6.59E-4
	1.0	1.02	6.41E-4
	1.5	1.42	3.87E-3
	2.0	1.72	3.62E-3
	2.5	1.99	8.89E-4

Table 13: Performance and the standard deviations of CP

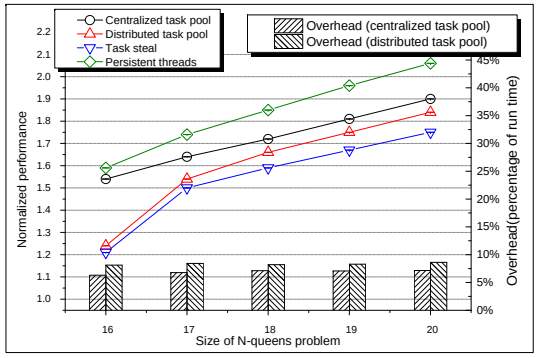


Figure 15: Performance comparison of N-queens solver (loading ratio = 20)

Scheme	Input	Perform.	SD
Centralized task pool	16	1.54	3.29E-4
	17	1.64	1.43E-4
	18	1.72	3.85E-5
	19	1.81	9.82E-6
	20	1.90	1.21E-6
Distributed task pool	16	1.24	1.09E-4
	17	1.54	1.08E-3
	18	1.66	2.12E-4
	19	1.75	5.31E-5
	20	1.84	3.74E-6

Table 14: Performance and the standard deviations of N-queens solver

8.2.1. CP

The normalized performance and the standard deviations of CP with four load balancing schemes are shown in Fig. 14. All load balancing schemes achieved better performance under larger granularity exponent except the persistent threads scheme. The centralized task pool approach outperforms all other schemes with a performance of 2.0 on average and up to 2.76. The persistent threads method has the worse performance of 0.96 on average, demonstrating that the overhead of this method outweighs the benefit for CP. Performance of the distributed task pool scheme is slightly better than the task steal scheme. The centralized or distributed task pool scheme adopts a static loading ratio of 16. The average overhead of the centralized task pool approach and the distributed task pool approach is 1.87% and 2.30%,

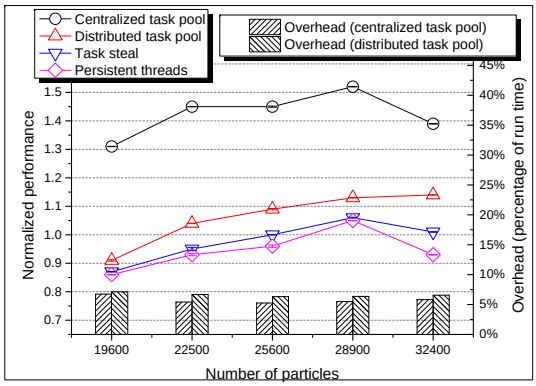


Figure 16: Performance comparison of potential distribution (loading ratio = 4)

Scheme	Input	Perform.	SD
Centralized task pool	19600	1.31	4.57E-4
	22500	1.45	7.78E-4
	25600	1.45	2.03E-3
	28900	1.52	5.93E-4
	32400	1.39	1.03E-3
Distributed task pool	19600	0.91	2.44E-3
	22500	1.04	6.47E-4
	25600	1.09	2.39E-3
	28900	1.13	7.90E-4
	32400	1.14	4.68E-4

Table 15: Performance and the standard deviations of potential distribution

respectively.

8.2.2. N-queens solver

As shown in Fig. 15, the persistent threads method achieves the best performance for N-queens solver, which is 1.84 on average and up to 2.06. The task steal scheme has the worse performance. The centralized task pool scheme obtains a performance of 1.72 on average and up to 1.90, which is better than the distributed task pool scheme. The centralized or distributed task pool scheme adopts a static loading ratio of 20. The average overhead of the centralized task pool approach and the distributed task pool approach is 6.87% and 8.33%, respectively.

8.2.3. Potential distribution

As shown in Fig. 16, the centralized task pool scheme achieves the best performance for potential distribution, which is 1.42 on average and up to 1.52. The distributed task pool scheme has a performance of 1.06 on average and up to 1.14. The task steal approach and the persistent threads scheme do not perform well for this application, showing a performance lower than 1.0 for most inputs. The centralized or the distributed task pool scheme adopts a static loading ratio of 4. The average overhead of the centralized task pool approach and the distributed task pool approach is 5.74% and 6.61%, respectively.

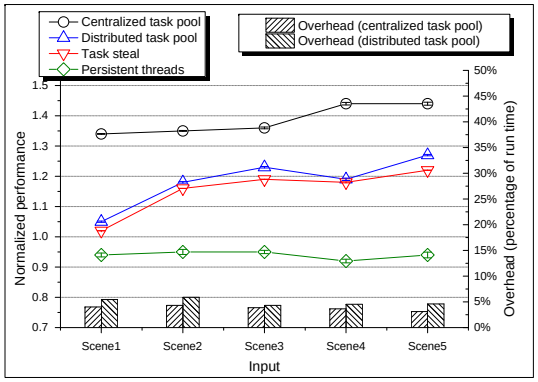


Figure 17: Performance comparison of ray-tracing (loading ratio = 6)

Scheme	Input	Perform.	SD
Centralized task pool	scene1	1.34	1.31E-3
	scene2	1.35	1.46E-3
	scene3	1.36	2.96E-3
	scene4	1.44	4.22E-3
	scene5	1.44	4.95E-3
Distributed task pool	scene1	1.05	2.11E-3
	scene2	1.18	2.51E-3
	scene3	1.23	2.48E-3
	scene4	1.19	3.95E-3
	scene5	1.27	2.25E-3

Table 16: Performance and the standard deviations of ray-tracing

8.2.4. Ray-tracing

From Fig. 17, we can see that the centralized task pool scheme achieves the best performance for ray-tracing, which is 1.38 on average with a peak at 1.44. The distributed task pool scheme achieves the second best performance of 1.18 on average and up to 1.27. The persistent threads scheme has the worse performance of 0.94 on average. The centralized or distributed task pool scheme adopts a static loading ratio of 6. The average overhead of the centralized task pool approach and the distributed task pool approach is 3.79% and 4.95%, respectively.

8.2.5. Summary

Overall, the centralized task pool scheme achieves the best performance for CP, potential distribution and ray-tracing. Although the persistent threads method obtains the best performance for N-queens solver, it has a poor performance of 0.95 on average for CP, potential distribution and ray-tracing. Therefore, the persistent threads scheme does not fit all irregular applications with thread-level load imbalance. Finally, the distributed task pool approach and the task steal approach have modest performance.

By observing the utilization graphs and the performance graphs of each application, we can see a correlation: each application has different GPU utilization under different input, and generally the highest performance is achieved on the minimum utilization. For example, CP has the lowest utilization of 29.5% among all applications, and at the same time achieves the highest performance of 2.76 with the centralized load balancing scheme.

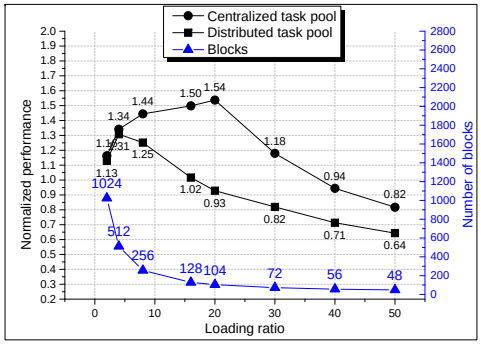


Figure 18: The impact of loading ratios on the performance of CP (granularity exponent = 1.0)

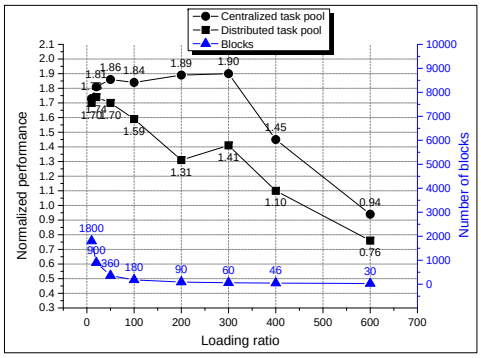


Figure 19: The impact of loading ratios on the performance of N-queen problem (N=19)

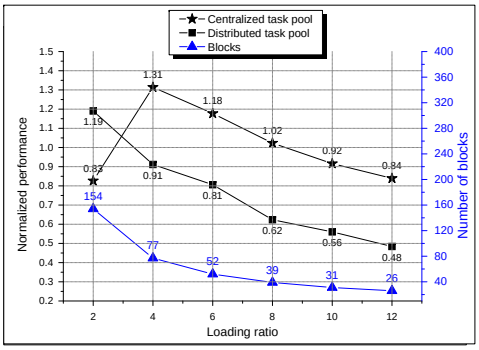


Figure 20: The impact of loading ratios on the performance of potential distribution (19600 particles)

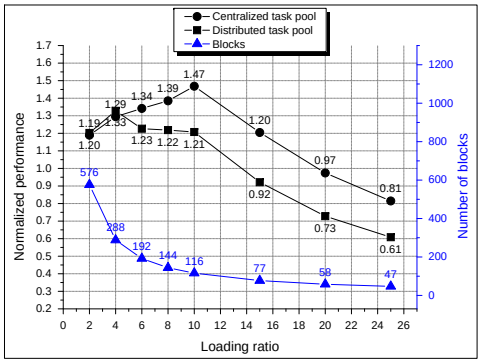


Figure 21: The impact of loading ratios on the performance of ray-tracing (scene3)

8.3. Evaluation of the adaptive loading ratio method

In the experiments in Sec. 8.2, the centralized or the distributed task pool approach works with a static loading ratio R for each application. As a result, applications cannot respond to the changes of inputs and fail to achieve the optimal performance. In this section, we first study the impact of loading ratios on performance. Then we compare the performance of the static loading ratios with that of the adaptive loading ratio method on different inputs.

8.3.1. Impact of loading ratios on performance

Fig. 18, Fig. 19, Fig. 20 and Fig. 21 show the impact of loading ratios on performance for all applications. Each application uses a fixed input with varied loading ratios. We define the *valid range* of loading ratios as a range of loading ratios that can keep the normalized performance of an application at or above 1.0 with a certain load balancing scheme. There are two important observations. First, N-queens solver has the widest valid range among all applications since it owns the highest parallelism (see Table 9). Similarly, CP has the second widest valid range since it possesses the second highest parallelism (see Table 8) Second, the curves of the centralized task pool approach of all applications match Fig. 6 in Sec. 4.4.1. That is, application performance keeps growing with the increasing of the loading ratios until there are insufficient blocks on the GPU. Therefore we can adopt the loading ratio that keeps just enough blocks on the GPU for good performance. This is the rationale of the adaptive loading ratio method.

8.3.2. Performance of the adaptive loading ratio method

Fig. 22, Fig. 23, Fig. 24 and Fig. 25 show the performance of the adaptive loading ratio method with the centralized task pool approach for all applications under different inputs. There are three observations. First, performance of the adaptive loading ratio method is better than that with static loading ratios in most cases. On average, the performance with static loading ratios is 2.00, 1.72, 1.42 and 1.38 for CP, N-queens solver, potential distribution and ray-tracing, respectively. With the adaptive loading method, the average performance becomes 2.01, 1.78, 1.46 and 1.42 for four applications, respectively. Actually the static loading ratios are already near-optimal thus it is hard to achieve any significant improvement. Second, the adaptive loading ratio method automatically derives different loading ratios for different inputs. For example, it derives loading ratios 7,8,10,11 and 12 for different scenes in Fig. 25. However, for CP and 17-queens to 20-queens, the method derives a same loading ratio for each application regardless of the inputs. The reason is that the two applications keep the same parallelism under different inputs(eg. the N-queens solver processes batches of 1152000 tasks for 17-queens and above. See Sec. 6.2.1). Third, the adaptive loading ratio method achieves similar performance with the optimal performance in Sec. 8.3.1. For example, Fig. 18 shows that the optimal performance of CP under a granularity exponent of 1.0 is 1.54 achieved at a loading ratio of 20. And Fig. 22 shows that the adaptive loading method derives a loading ratio

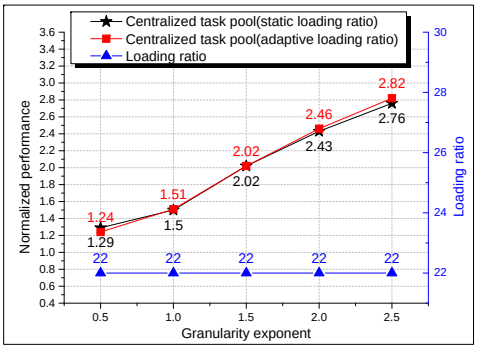


Figure 22: Performance of CP with the adaptive loading ratio method

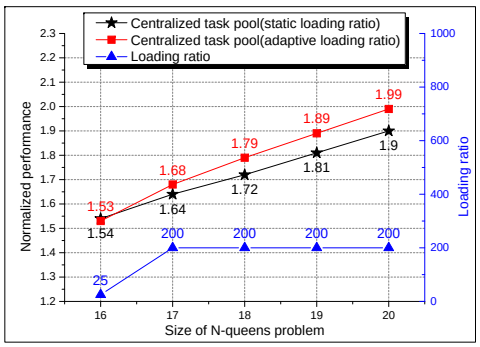


Figure 23: Performance of N-queens solver with the adaptive loading ratio method

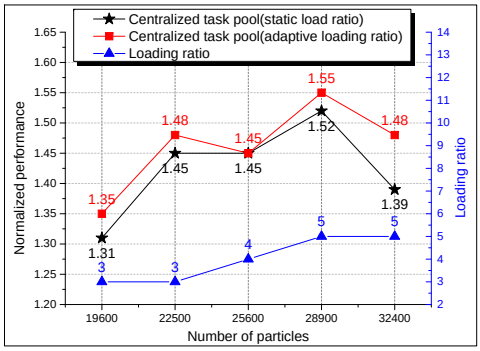


Figure 24: Performance of potential distribution with the adaptive loading ratio method

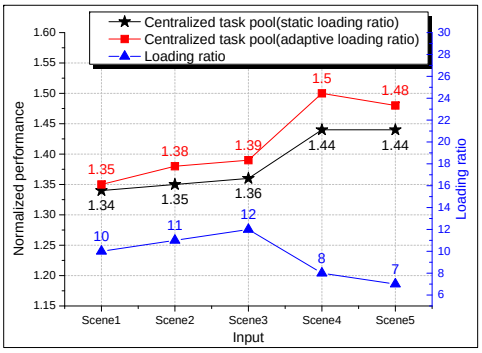


Figure 25: Performance of ray-tracing with the adaptive loading ratio method

of 22 and achieves a performance of 1.51.

8.3.3. Summary

In summary, the adaptive loading ratio method works effectively and outperforms static loading ratios in most cases. Evaluation results manifest that the performance curve of each application depends on its parallelism, utilization/irregularity and the loading ratio. The results also suggest that generally we should stop increasing the loading ratio to keep enough blocks on the GPU for good performance.

9. Conclusion

In this work, we introduce an open-source library called CUIRRE for reducing thread-level load imbalance of irregular applications with the task pool approach. Besides load balancing, the library can also characterize the irregularity, GPU utilization and thread granularity of irregular applications.

We characterized and optimized four applications with the library. Among all applications, CP has the highest irregularity of 3.41 and the lowest utilization of 29.4%, and achieves the most significant performance improvement of 2.76× with the centralized task pool approach. Besides, N-queens solver has the best parallelism while potential distribution has the coarsest average thread granularity. We are able to highlight a strong correlation between the utilization, parallelism and achievable performance improvement with the library: applications with lower GPU utilization and higher parallelism can achieve a larger performance improvement.

The performance of the centralized task pool and the distributed task pool in the library is compared with the task stealing approach and the persistent threads approach from existing work. Overall, the average normalized performance for the centralized task pool approach, the distributed task pool approach, the task stealing approach and the persistent threads approach is 1.63, 1.31, 1.25 and 1.17, respectively. We conducted theoretical analysis as well as experimental measurement on the overhead of the modules in the library. The average overhead for the load balancing module and the diagnostics module is 5.06% and 1.39%, respectively. The Centralized task pool approach outperforms all other schemes for its more balanced load and smaller overhead. If applying the adaptive loading ratio method, the performance can be further improved.

Acknowledgment

The authors would like to thank Linghe Kong, Xiaoyang Liu, Manuel Charlemagne and anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This research is supported by NSF of China under grant No.60773091, No.61100210, Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), China, and Science and Technology Committee of Shanghai Municipal under grant No.12dz1507400.

[1] Nvidia corporation, the ray-tracing engine, CUDA Community Showcase, 2013, http://www.nvidia.cn/object/cuda-apps-flash-new.html.

[2] E. Z. Zhang, Y. Jiang, Z. Guo, X. Shen, Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping, in: Proc. of the 24th ACM International Conference on Supercomputing(ICS), 2010, pp. 115–126.

[3] T. Zhang, W. Shu, M.-Y. Wu, Optimization of n-queens solvers on graphics processors, in: 9th International Symposium on Advanced Parallel Processing Technologies(APPT), 2011, pp. 142–156.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, Analyzing cuda workloads using a detailed gpu simulator, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009, pp. 163 – 174.

[5] Nvidia corporation, nvidia cuda campus design & implement contest, 2009, http://cuda.csdn.net/contest/pro/index.aspx.

[6] S. Solomon, P. Thulasiraman, Performance study of mapping irregular computations on gpus, in: IEEE International Symposium on Parallel & Distributed Processing, 2010, pp. 1–8.

[7] S. Frey, G. Reina, T. Ertl, Simt microscheduling: Reducing thread stalling in divergent iterative algorithms, in: 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2012, pp. 399–406.

[8] D. Cederman, P. Tsigas, On dynamic load balancing on graphics processors, in: 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008, pp. 57–64.

[9] D. Cederman, P. Tsigas, Dynamic load balancing using work-stealing, Wen-mei W. Hwu (Eds.), GPU Computing Gems Jade Edition, Morgan Kaufmann (2011) 485–499.

[10] T. Aila, S. Laine, Understanding the efficiency of ray traversal on gpus, in: Proceedings of High Performance Graphics, 2009, pp. 145–149.

[11] S. Tzeng, A. Patney, J. Owens, Task management for irregular-parallelworkloads on the gpu, in: High Performance Graphics, 2010, pp. 29–37.

[12] L. Chen, O. Villa, S. Krishnamoorthy, G. Gao, Dynamic load balancing on single- and multi-gpu systems, in: IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1–12.

[13] H. Jia, Y. Zhang, W. Wang, J. Xu, Accelerating viola-jones facce detection algorithm on gpus, in: IEEE 9th International Conference on High Performance Computing and Communication, 2012, pp. 396–403.

[14] P. Beck, M. Nehmeier, Parallel interval newton method on cuda, in: 11th International Conference on Applied Parallel and Scientific Computing, 2012, pp. 454–464.

[15] H. Wu, G. Diamos, S. Li, S. Yalamanchili, Characterization and transformation of unstructured control flow in gpu applications, in: First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems, 2011.

[16] M. Rehman, Exploring irregular memory access applications on the gpu, master thesis, International Institute of Information Technology, 2010, http://www.contrib.andrew.cmu.edu/ suhailr/thesis.pdf.

[17] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on gpus, in: IEEE International Symposium on Workload Characterization (IISWC), 2012, pp. 141–151.

[18] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, Nvidia tesla: A unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.

[19] C. Cuda, Programming guide, NVIDIA Corporation (May 2014).

[20] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, X. Liang, An energy-efficient and scalable edram-based register file architecture for gpgpu, in: International Symposium on Computer Architecture (ISCA), 2013.

[21] S. Z. Gilani, N. S. Kim, M. J. Schulte, Power-efficient computing for compute-intensive gpgpu applications, in: High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, IEEE, 2013, pp. 330–341.

[22] W. W. Fung, I. Singh, A. Brownsword, T. M. Aamodt, Hardware transactional memory for gpu architectures, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 296–307.

[23] S.-Y. Lee, C.-J. Wu, Characterizing the latency hiding ability of gpus, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2013.

[24] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, W. M. Hwu., Parboil: A revised benchmark suite for scientific and commercial throughput computing, IMPACT Technical Report, http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf (2012) 1–12.

[25] A. Bozinovski, S. Bozinovski, n-queenss pattern generation: an insight into space complexity of a backtracking algorithm, in: international symposium on Information and communication technologies, 2004, pp. 281 – 286.

[26] S. Khan, M. Bilal, M. Sharif, M. Sajid, R. Baig, Solution of n-queen problem using aco, in: IEEE 13th International Multitopic Conference(INMIC), 2009, pp. 1–5.

[27] R. G. Spallek, T. B. Preuber, B. Nagel, The world record by fpgas, QUEESNTUD project report, http://queens.inf.tu-dresden.de/ (2009) 1–1.

[28] R. Y. Rubinstein, D. P. Kroese, Simulation and the monte carlo method, second eds., John Wiley & Sons (2007) 1–372.

[29] Nvidia corporation, nvidia's next generation cuda compute architecture: Fermi, Nvidia White Paper, http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (2009) 1–22.

[30] Nvidia corporation, nvidia cuda sdk 4.2, NVIDIA CUDA software download, 2012, https://developer.nvidia.com/cuda-toolkit-42-archive.