Abstract

Heterogeneous computing is an exciting new field that aims to make use of all processors on a given architecture in order to perform general-purpose computing as efficiently as possible. However, designing a programming framework to do this that is fast, portable, and responsive to problem behavior, while being easy to program and introducing minimal overhead, has proven challenging for many reasons. DICE is one such system that has shown promise; here, I show that DICE can effectively load-balance multiple different types of workloads among different device types, and adapt to irregularity in work runtimes.

1 Introduction

As the increase in the computing power of modern microprocessors has slowed in recent years, more research effort has been devoted within the computer science community to exploring parallel architectures and algorithms. A relatively new field that is garnering attention is heterogeneous computing, in which general-purpose computing is done on an architecture that includes multiple different types of processors. Conventional wisdom dictates that writing a program for a specific device with specialized capabilities that best serve the work required by the given problem is the most efficient way to solve it; however, researchers are coming to accept that for certain tasks that entail generic computation, a heterogeneous solution is best. Though exploiting the availability of processing power across multiple devices promises
to deliver speed improvements, it presents a number of challenges to the programmer. To understand these challenges, we first need to understand a couple of concepts fundamental to heterogeneous computing.

One such concept related to analyzing parallel programs is grain size or granularity. The grain size is essentially the extent to which a problem is broken up into smaller sub-problems, which can be solved in parallel. Programs with grain sizes that do not change are said to be static, while those with grain sizes that change over time, either according to a certain predefined manner or as calculated during the runtime of the program, are called dynamic. Another facet of analyzing problems is the question of irregularity: if sub-problems of similar size result in similar workloads, a problem is said to be regular; if a sub-problem’s size has little correlation with its runtime on a particular architecture, the problem is irregular.

Choosing the grain size for a program solving a particular problem must be done carefully; too large a grain size results in an imbalance in the work distribution among devices or threads, which could result in device idle time, while too small a grain size risks extra overhead in device synchronization and data transfer. Furthermore, different microprocessors with different speeds and designs may have different optimal grain sizes for different problems; coordinating the best grain sizes for a given algorithm across multiple devices within a single architecture is an imperative in heterogeneous computing. Finally, settling on a specific granularity is difficult because problem irregularity can make sub-problems unpredictable in their runtimes.

Another issue that all heterogeneous system programmers must face is how to distribute the work necessary to solve a given problem across all of the processors in a given architecture. While picking a grain size determines how large a sub-problem each device will solve, coordinating the amount of work that each device does so as to minimize system idle time is a different challenge. Unless a problem is very regular, choosing a static work distribution for each device will prove inefficient; a dynamic method of scheduling work on each device would serve much better here, as it can respond and adapt to different instances of a problem.

Heterogeneous programming models all strive to address these issues while presenting a simple interface for their users. Ideally, one could port an algorithm to a heterogeneous programming platform without needing an in-depth knowledge of parallel programming constructs or architecture specifics. Providing an architecture-independent framework is another big goal of heterogeneous computing frameworks.
Previous work has attempted to address one or more of these problems. TBB [6] and OpenMP [5] can automatically shrink the grain size over time linearly, but do not adapt well to irregular problems; most other platforms require choosing a static grain size before runtime. This can take weeks of the programmer’s time to identify a good grain size for a given problem, which can only be used on a specific architecture; also, it may be difficult to find a single grain size that works well on all processors of an architecture, and most current systems do not allow for specifying separate grain sizes for each device type. To address load-balancing in a heterogeneous environment, some systems feature a form of automatic or dynamic load-balancing, a necessary component for handling irregular problems. In terms of programming ease, systems widely vary in the responsibilities they place upon the programmer. Some frameworks require written code for each device type [7], while others do not, and each programming framework places different requirements on the extent to which the programmer must deal with memory management and data transfer between devices.

The DICE (Dynamic Irregular Computing Environment) programming framework [8] provides a simple interface for writing heterogeneous parallel algorithms that are architecture-independent and can adaptively modify the grain size used for each device, as well as the distribution of work across different device types. The interface requires a programmer-provided method of breaking a task into sub-tasks; this is usually much more tractable than selecting a good granularity for a given problem and architecture, since the programmer can be expected to have a deep understanding of the algorithm, but not necessarily of the specifics of the hardware platform it is running on. DICE’s design is especially important for irregular tasks, in which estimating the work needed to complete each sub-problem can be difficult or instance-specific. In this paper, I investigate DICE’s ability to dynamically assign granularity and perform load-balancing for an animated path-tracer.

Section 2 introduces the DICE programming framework. Section 3 describes the general path-tracing algorithm, object animation, and the relevant data structures. Section 4 details my implementation of the animated path-tracer, and section 5 lists the runtime results of rendering several scenes on one architecture. Section 6 offers ideas for future work.
2 The DICE framework

DICE is a C++ programming framework for dynamically modifying the granularity at which a parallel task is broken into sub-tasks and distributed among devices of different amounts of computing power. To use it, the programmer defines a parallelizable problem as a DICE work task with a constructor and two key functions: `execute()` and `dice()`. The constructor delineates what data the algorithm will operate on, and where the computed result should be stored. The `execute()` function expresses how a sub-problem is actually solved. It accesses the data members of the work task initialized within the constructor. The `dice()` function takes as input a vector of non-negative floating-point numbers whose sum is 1. This function must return a vector of sub-tasks, where the expected runtime of each sub-task is proportional to its corresponding floating-point number in the input vector (see figure below). Using this framework, the programmer can define how to divide a given problem into sub-problems for a given grain size, without having to choose a specific one or implement a method for calculating it.

When a DICE task is submitted for the first time, DICE has no information about how well the task will perform on each device, and so it distributes the work equally among all devices in the architecture. Each device distributes its sub-task equally among all of its threads; the grain size is effectively the amount of work assigned to a device, divided by the number of available threads. After a DICE task has finished, the DICE framework examines the time it took for each device to finish its assigned sub-task, and updates its internal model of which processors best solve that problem. It uses this information to estimate a better workload distribution in order to

\[
\text{Work task} \quad \{0.2, 0.5, 0.3\}
\]

\[
\begin{array}{ccc}
0.2 & 0.5 & 0.3 \\
\end{array}
\]

Figure 1: Illustration of the `dice()` function. Each sub-task returned should require an estimated proportion of the total runtime that the parent task would have taken.
minimize device idle time during future executions of the same task. In doing this, DICE automates the regulation of workload distribution among devices of different computing capacities. It also maintains information about what grain size works best for each device type in the architecture. This allows it to adapt to any heterogeneous architecture, as well as irregularity in the problem itself, while adding little overhead to the original program. It was with these goals in mind that we chose path-tracing to test the abilities of DICE.

3 Path-Tracing

Path-tracing is a decades-old method of rendering a scene that simulates real-life light transport from light sources to the human eye [2]. In the real world, the perception of an object in three-dimensional space is the result of light reflecting off of one or more objects and hitting the retina. Path-tracing performs the inverse of this: light rays are generated that start at the camera lens and project away from the camera into the three-dimensional world, called primary rays. When a primary ray intersects with an object, its resulting color is calculated using the object’s material properties, and the amount of light that hits the object at the point of intersection. This requires calculating an integral over every direction that light can come from and hit the object surface; the integral is approximated using a Monte Carlo method, where the average of random samples within the integral bound is taken [1]. Once a color for a primary ray has been determined, it is assigned to the corresponding pixel in the final rendition of the scene. Rays are also recursively traced from the point of intersection of a primary ray with an object to account for reflection off of and refraction within objects. The path-tracer that is the subject of this paper, in addition to tracing primary rays, also casts rays for producing reflections on object surfaces.

While the path-tracing algorithm for rendering a three-dimensional scene as a two-dimensional image is relatively easy to understand and implement, its speed decreases quickly as the scene complexity grows. The brute-force approach to path-tracing performs an intersection test between every generated ray and every face of every object in the scene; performing a single ray-face intersection is a fast operation, but doing this for millions of rays and thousands of object faces entails a lot of unnecessary work. To improve performance, path-tracers frequently employ ray-surface intersection accel-
eration data structures, which reduce the number of object faces each ray must test for intersection.

One such acceleration structure is a bounding-volume hierarchy (BVH), a tree in which each node represents a collection of object faces in the scene, and keeps track of the smallest axis-aligned bounding box that contains all of the object faces in the node. The bounding box of a non-leaf node must contain the bounding boxes of all of its children; the bounding box at the root of a BVH contains every object face in the scene. After constructing a BVH, when a ray is cast into the scene, it is tested for intersection with the bounding box at the root of the tree. If the ray passes through the bounding box, it tests for intersection with the node’s children; if it does not hit the bounding box, then it is guaranteed that the ray does not intersect with any object face in the node. When the BVH traversal reaches a leaf node, the ray is tested for intersection with all of the object faces in the leaf node. Using a BVH vastly reduces the number of ray-face intersection tests that must be done when tracing each ray.

Originally, path-tracing was conceived in order to render static scenes — ones in which the objects do not move. Rendering dynamic scenes is much harder; during every frame, the objects in the scene move to new positions, and so the BVH is no longer an accurate representation of the scene and must be reconstructed. Furthermore, object animations must be stored in some manner, and should be quickly accessible in order to update the positions of each object during every frame. To do this, I implemented a simple scene graph, another tree-like data structure in which nodes store affine transformations and references to objects in the scene [4]. To animate a scene object, a scene graph traversal starts at the root; as it traverses nodes in subsequent levels of the tree, it pre-multiplies its current transformation matrix by the transformation stored at each node. This accumulates transformations that will be applied to the scene object, stored at a leaf of the scene graph. To represent different animations across multiple frames, each scene graph node stores an array of transformations, one for each frame. Using a scene graph allows for efficient re-use of transformations that must be applied to multiple objects in the scene, and enables object transformations to be defined relative to other scene objects.

Once a bounding-volume hierarchy has been constructed for a scene, tracing primary and secondary rays to produce an image is a task that is easily parallelized; each ray can be traced separately, reading from shared data structures. However, path-tracing an animated scene is highly irregular in
both space and time: calculating the colors of different pixels in the image can take wildly varying amounts of time, depending on what objects in the scene each pixel’s associated primary and secondary rays hit, and calculating the color of the same pixel can take different amounts of work each frame as scene objects move to different locations in the scene. Therefore, choosing a static grain size is extremely difficult and impractical, as the workload for computing each pixel’s color depends on the objects in the scene, their animations, and the structure of the BVH calculated each frame. For these reasons, animated path-tracing was an ideal problem for evaluating the effectiveness of the DICE platform in dynamically choosing grain sizes and assigning workloads to devices with different capabilities.

4 Implementation

The animated path-tracer takes as input two files: the first follows the OBJ file format for defining a three-dimensional scene, and the second describes the animations to apply to each object in the scene. After parsing the OBJ file, the path-tracer stores each scene object as a vector of vertices, vertex normals, and surface materials. A vector of faces is also kept, which consists of tuples of offsets into the vertex vector; each tuple represents one object face. The second file is parsed to construct the scene graph of animations and scene objects; the transformations that will be applied to each object on every frame are computed here. For the remainder of the program’s runtime, this scene data will remain unchanged and in the same place in memory.

Next, the path-tracer generates the BVH tree structure that will be used every frame. The tree structure is created as follows: at the root node, all scene objects are included, and every child node in the BVH structure contains half of the scene objects of its parent node. The choice of which objects from the parent node to include in a child node is arbitrarily decided based on the order in which scene objects are defined in the OBJ file, a limitation of this implementation that in most cases will incur a performance penalty when rendering, but allows for parallelizing the re-population of the BVH every frame. This process is continued until all leaf nodes contain exactly one scene object. Then, a BVH of limited size is built for each scene object separately, using the surface-area heuristic; this algorithm uses a greedy, top-down approach that recursively partitions nodes by minimizing the expected sub-tree traversal runtimes [3]. These object-specific BVHs are
in local coordinate-space, and only need to be calculated once, since the scene objects do not move relative to their local coordinate frames. Once this operation is completed, every scene object has its own BVH, which allows for efficient ray-intersection testing with the faces of that object.

After finishing this preliminary work, the program enters the main rebuild-render cycle. In this cycle, two DICE tasks are submitted every frame: the BVH rebuild task and the shading task. The BVH rebuild task populates the BVH for the current animation frame with the bounding boxes of all of the objects in the scene, and the bounding boxes of each object-specific BVH. It does this by applying each scene object’s corresponding transformation in the scene graph to all of the bounding boxes in the object-specific BVH for that object, which transforms them from local coordinates to world coordinates, then combining these bounding boxes to populate the general BVH, in which nodes may contain more than one scene object. Doing this creates a BVH for the scene in world coordinates while retaining all of the mesh geometry in local coordinates. This is done so that, when tracing rays and performing ray-object intersection tests, the path-tracer can apply the inverse transformation of a scene object to the ray before performing the intersection test, and so avoid having to transform the vertices of every scene object every frame. Each bounding box needed for constructing the BVH can be computed in parallel, so sub-tasks are created by splitting the BVH into subsets of nodes. Since the execute() function consists of combining a fixed number of bounding boxes and applying a transformation to the result, this task is very regular in its behavior.

The shading task generates primary rays for each pixel in the final image, calculates their colors by traversing the BVH and casting secondary rays as necessary, and saves the result to a vector. Sub-tasks are created by splitting the screen into subsets of pixels; the smallest possible sub-task is computing the color of one pixel by casting as many primary and secondary rays as are needed. As previously described, this task can be very irregular, depending on the characteristics of the scene and the animation applied to the scene objects.

The pseudo-code for the main rebuild-render cycle is shown below. Every frame, the camera’s location and orientation is updated according to its defined animation, and all of the primary rays for that frame are generated. Then, the program waits for the BVH build for the previous frame to finish before asynchronously submitting the task for building the BVH for the current frame to DICE, to ensure that no more than one BVH build is in flight.
at any given time. After the BVH build task has been submitted, the program waits for the shading task from two frames ago to finish, then submits the shading task for the previous frame. This is done because the shading task for the previous frame cannot be submitted until the BVH build for the previous frame has finished. In essence, the rebuild-render cycle is pipelined, so that the BVH rebuild for one frame is being done at the same time as the shading for the previous frame. Once a frame has been shaded, it is displayed on-screen.

```c
// Input n: the animation frame number to render
function rebuild_render(n) {
  update position and orientation of camera
  generate primary rays from updated camera

  while (bvh_build(n - 1) is not finished) {}
  DICE_submit(bvh_build(n))

  while (shading(n - 2) is not finished) {}
  DICE_submit(shading(n - 1))
  update_display(n - 2)
}
```

5 Results

I ran the animated path-tracer on an architecture consisting of two Intel Xeon E5 processors, each containing ten cores and running at 2.8 GHz, and one Intel Xeon Phi Coprocessor, with sixty cores running at 1.053 GHz. The animated path-tracer was tested on two scenes.

5.1 Cornell Box

The first scene is the famous Cornell box scene, which consists of two translucent spheres contained in a box with a single ceiling light (Figure 1). The scene includes 1,116 vertices and 2,188 faces. I added animation to the scene so that the two spheres bob back and forth between the walls of the scene, and the camera moves in a circle while facing the center of the box. The animation cycles every 100 frames, and the path-tracer was run for 500 frames. The results are shown below.
Figure 2: The famous Cornell box scene.

Figure 3: Path-tracer runtime for both tasks when run solely on the CPU. The BVH rebuild task runtimes are insignificant in comparison to the tracing task runtimes, and so are not visible in the graph.
Figure 4: Path-tracer runtime for both tasks when run solely on the GPU.

Figure 5: Path-tracer runtime for both tasks when run in heterogeneous mode.
Figure 6: The workload distribution between the CPU and the GPU for the BVH rebuild task.

Figure 7: The workload distribution between the CPU and the GPU for the tracing task.
The results for running the path-tracer on the CPU and the GPU separately are fairly uninteresting: the highly-regular BVH rebuild task takes a small fraction of the time that the tracing task does, and the tracing task sees periodic performance increases and decreases on each device as the portion of the scene being rendered changes. Note that the peaks and valleys of the tracing task runtimes on the CPU and the GPU do not overlap, demonstrating that each is better suited for handling certain scene arrangements in the animation.

When run in heterogeneous mode, the BVH rebuild task is quickly relegated only to the CPU; for a scene with so few objects, the time required to transfer any part of the BVH rebuild task to the GPU and coordinate collecting its results is larger than the time it takes for the CPU to complete the task in its entirety. The periodic spike in the BVH rebuild task runtime is a small error in the way DICE tracks task finish times; this was likely due to the fact that DICE was managing multiple task types running concurrently, a first for the DICE system.

With the tracing task, DICE on average assigns 25% of the task to the GPU. Examining the runtimes of the tracing sub-tasks on both devices, we notice that when the GPU finishes its assigned sub-task before the CPU,
DICE assigns a larger percent of the workload for the tracing task to the GPU on the following frame, and vice-versa; this indicates that DICE is reacting to irregularity in the tracing task as the scene animation progresses, and attempting to balance the workload so that each device finishes its sub-task at approximately the same time. Finally, we can also see that the total time to rebuild the BVH and perform path-tracing each frame is smallest when the path-tracer is run heterogeneously, proving that using DICE to perform automated load-balancing incurs performance gains for this particular problem.

5.2 Head with Moving Camera

The second scene is a three-dimensional model of a scanned human head. It contains 8,844 vertices and 8,842 faces. In the animation, the head sits under a square light as the camera passes in front of it. The camera never changes its orientation, so at the beginning and end of the animation, almost nothing appears on-screen; as the head enters and leaves the camera’s view, the work required to perform path-tracing changes quickly, making the animation a good candidate for testing DICE’s ability to adapt to changing scene dynamics. The animation repeats every fifty frames, and the path-tracer was run for 500 frames. Results are shown below.
Figure 9: Path-tracer runtime for both tasks when run in heterogeneous mode.

Figure 10: The workload distribution between the CPU and the GPU for the tracing task.
Figure 11: The tracing sub-task runtimes on each device when running the path-tracer in heterogeneous mode.

Figure 12: The tracing sub-task runtimes on each device for the last 100 frames, shown in better detail.
Results for the workload distribution for the BVH rebuild task are omitted because they are similar to that of the Cornell box scene. In this scene, although the amount of work needed to render each frame varies greatly throughout the animation, each device’s speed per unit of work changes much less. Because of this, the balance of work between the CPU and the GPU stays relatively constant throughout the animation, stabilizing after about twenty-five frames and only fluctuating within a two-percent bound thereafter.

6 Conclusion

I have demonstrated that the DICE programming framework is sufficiently robust to handle the highly irregular nature of performing animated path-tracing. It successfully adapts to peculiarities in tracing scenes with animations, and converges to a workload distribution that minimizes idle time by taking into account the speed and characteristics of each device. It was also easy to change the path-tracer to integrate it with DICE, a testament to DICE’s simplicity of use. Hopefully, these results will extend to integrations of DICE with even more complex problems and larger architectures.

7 Acknowledgements

I would like to thank my two advisors, Donald Fussell and João Barbosa, for their endless patience in giving me guidance throughout the duration of this project. That I was able to complete it with no prior experience working with heterogeneous systems is a testament to their knowledge. I would like to give special thanks to João for always being available to answer my questions about the DICE framework. I would also like to thank my other readers, Calvin Lin and Etienne Vouga, for their efforts to evaluate my work and provide feedback.

References


