A CUDA Runtime Target for the Sequoia Compiler

Michael Bauer, John P. Clark, Eric Schkufza, Alex Aiken
Stanford University
http://sequoia.stanford.edu

The Sequoia Programming Model

The Sequoia programming model is designed for writing locality-aware, portable parallel programs for deep memory hierarchies. In order to accomplish this, Sequoia presents an abstract machine model to the programmer that is a tree of memories (Figure 1). The programmer has no specific information about this tree of memories (i.e. depth or branching factor), but can assume that each level is progressively smaller and contains more powerful processing elements as one gets closer to the leaves.

This abstraction is capable of representing machines with many different levels of memory; everything from a desktop PC to clusters of SMPs with Cell or GPU accelerators (Figure 2). In addition, because the program is written for an abstract memory hierarchy the program is easily portable to machines without any source code modifications.

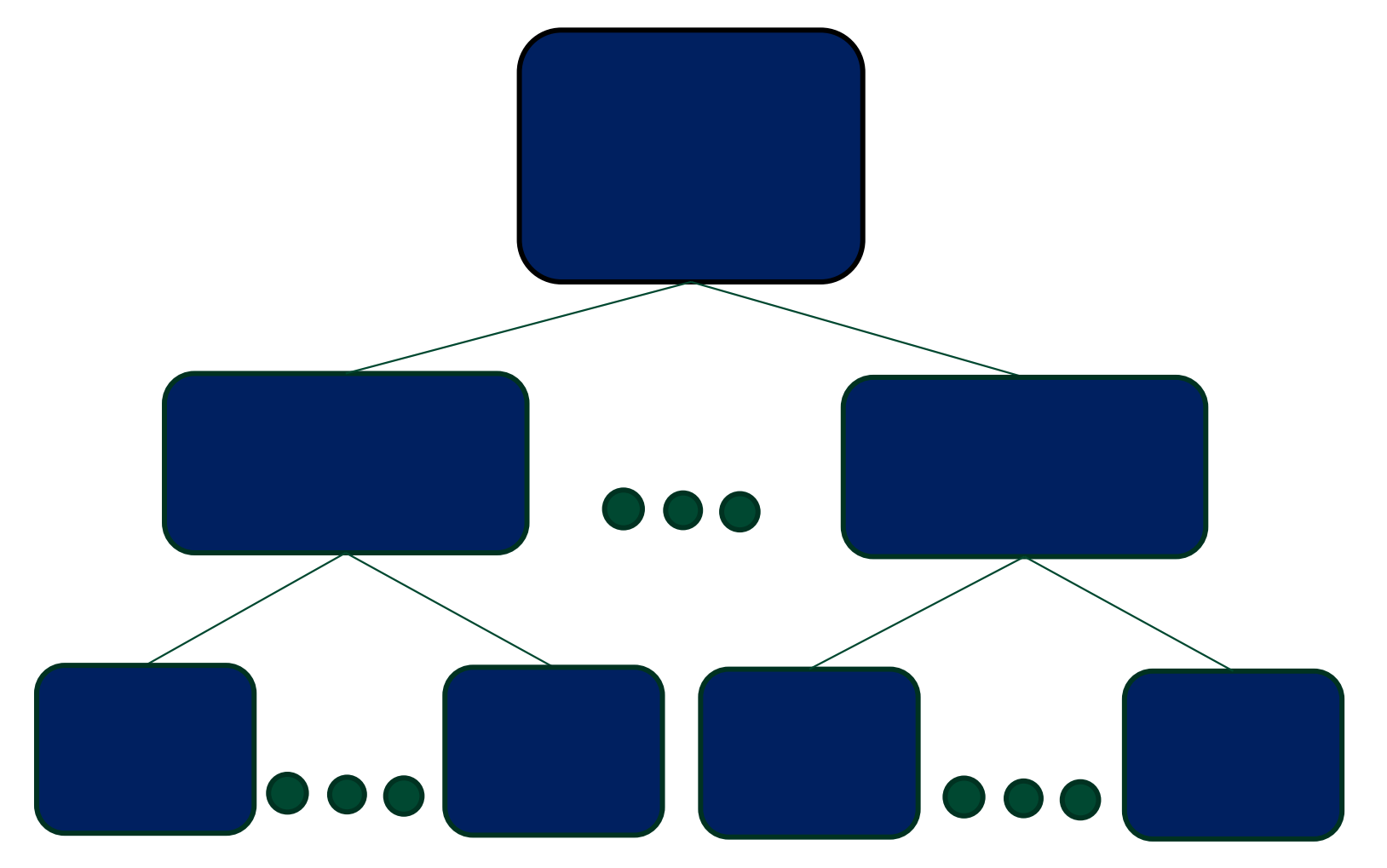
Figure 1. Sequoia Abstract Machine Model

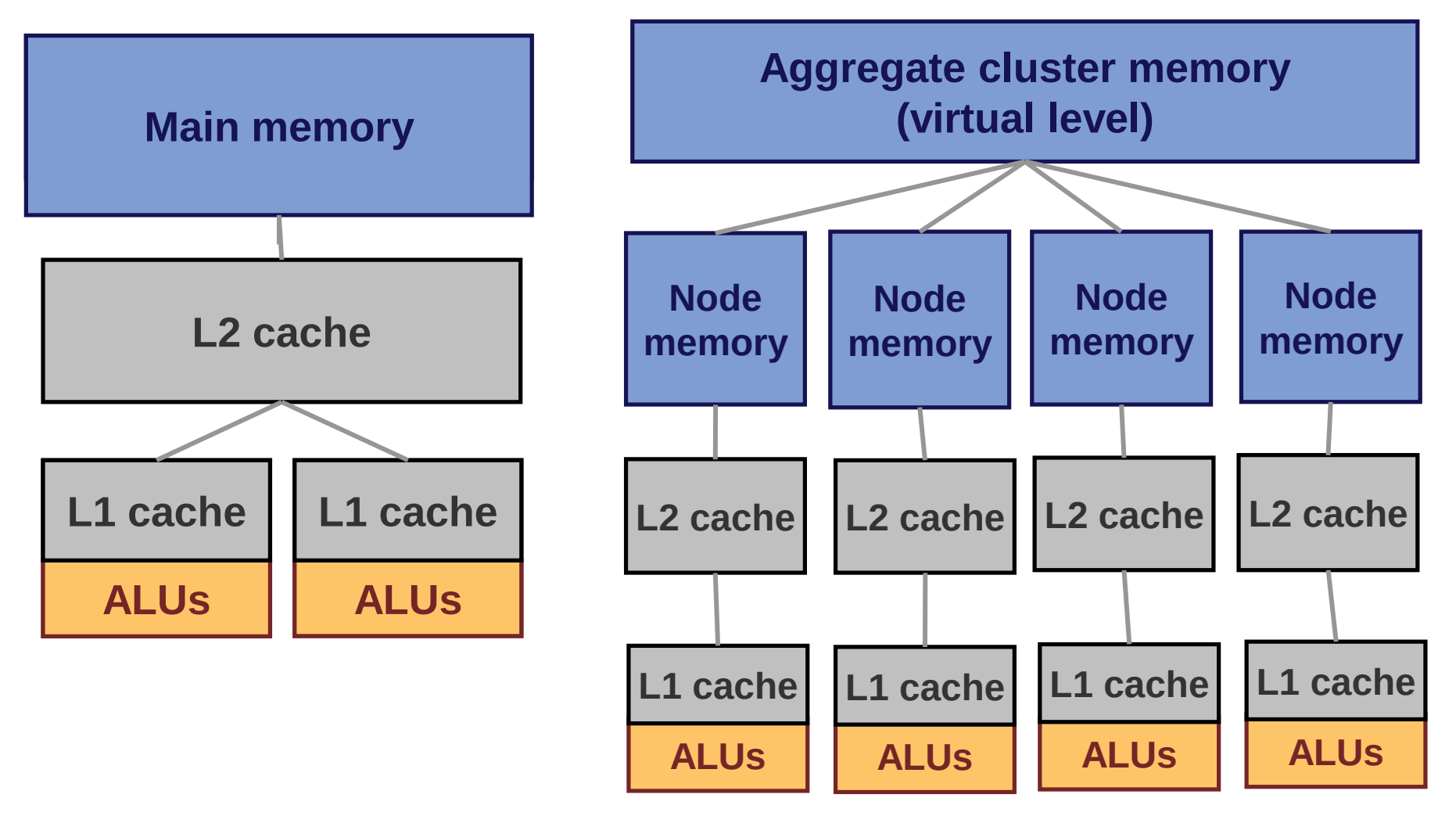

Figure 2. Capturing a dual-core CPU and cluster of CMPs in the Sequoia machine model

Mapping Sequoia onto CUDA

In order for the Sequoia compiler to be able to target GPUs, we have to be able to map the Sequoia programming model onto the CUDA programming model. This requires that all the features of the Sequoia language can be expressed in CUDA. Figure 4 gives a simple illustration of some of the important differences between Sequoia and CUDA.

When targeting Sequoia to CUDA, we ignore the CUDA abstract machine model and instead allow Sequoia to leverage its knowledge of the program to target the device directly. We therefore only launch as many cooperative thread arrays (CTAs) as there are streaming multiprocessors (SMs) on the device. The Sequoia compiler is then able to map tasks directly onto the SMs rather than simply launching CTAs and trusting the hardware to schedule them efficiently.

	CUDA	Sequoia
Types of Parallelism	- Data parallelism - Task parallelism	- Data parallelism - Task parallelism - Nested parallelism
Synchronization Mechanisms	- Intra-CTA - Across kernels	- Synchronous task launching
Memory Latency Hiding	- Interleave fine-grained data parallel comp.	- Overlap task execution and communication

Figure 4. A comparison between the Sequoia and CUDA programming models

Targeting Multiple GPUs

A major challenge with CUDA currently is programming multi-GPU systems. Sequoia is easily able to handle multiple GPU systems since its runtimes easily compose. Using a CMP runtime Sequoia is able to launch multiple threads on the CPU (one for managing each GPU). The CMP runtime easily composes with the GPU runtime enabling Sequoia programs to run on systems with multiple GPUs. Sequoia can even place a cluster runtime on top of the CMP runtime to make it possible to target large MPI clusters with multiple GPU accelerators per node.

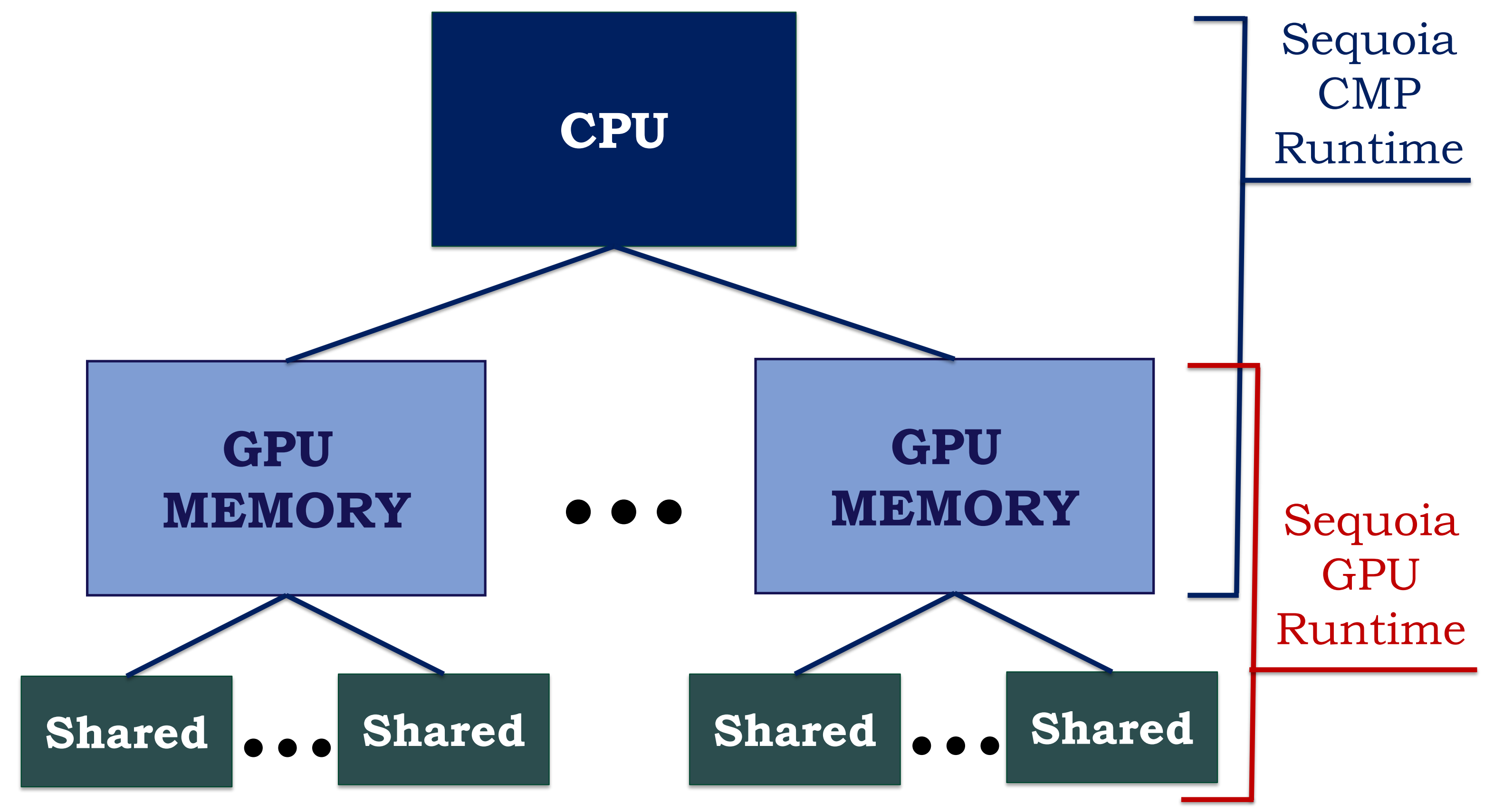

Figure 6. Sequoia representation of a multi-GPU system

Programming in Sequoia

The primary unit of computation in Sequoia is a *task*. Tasks are pure functions whose entire working set must be described by the arguments to the task. Tasks provide isolation from other tasks and also provide locality information to the compiler by describing their working set.

There are two types of tasks:

- *inner*: these tasks describe how to partition data and work into sub-tasks
- *leaf*: these tasks perform computation

It is the goal of the programmer to map tasks onto the memory hierarchy.

Tasks are parameterized using *tunable* variables that are specified in a special mapping file at compile-time. This enables tasks to be portable across different architectures.

```
void task<inner> matmul( in    float A[M][P],
                         in    float B[P][N],
                         inout float C[M][N] )
{
  // Code to name submatrices of A, B, and C
  // called Ablks, Bblks, and Cblks, respectively
  // block sizes are given by U, V, and X

  // Compute all blocks of C in parallel.
  mappar (int i=0 to M/U, int j=0 to N/V) {
    mapseq (int k=0 to P/X) {
      // Invoke the matmul task recursively
      // on the subblocks of A, B, and C.
      matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
    }
  }
}
void task<leaf> matmul( in    float A[M][P],
                        in    float B[P][N],
                        inout float C[M][N] )
{
  // Compute matrix product directly
  for (int i=0; i<M; i++)
    for (int j=0; j<N; j++)
      for (int k=0; k<P; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

Figure 3. Example matrix multiplication in Sequoia

Eliminating Host-Device Copies

One advantage of the Sequoia programming model is task working sets are explicit. By coupling this information with the compiler's knowledge of where tasks are scheduled to be run in the memory hierarchy, the compiler is able to create an intermediate representation rich enough to perform copy elimination. Since Sequoia deals with bulk transfers of data rather than individual variables, every eliminated copy results in significant performance gains.

In the context of CUDA, this is most noticeable when dealing with copies between host and device memories. Sequoia is able to reason about reuse of data across kernel calls and thereby avoiding moving data unnecessarily between the host and device. There are several different copy elimination patterns that the Sequoia compiler is capable of recognizing, one of which is illustrated in Figure 5.

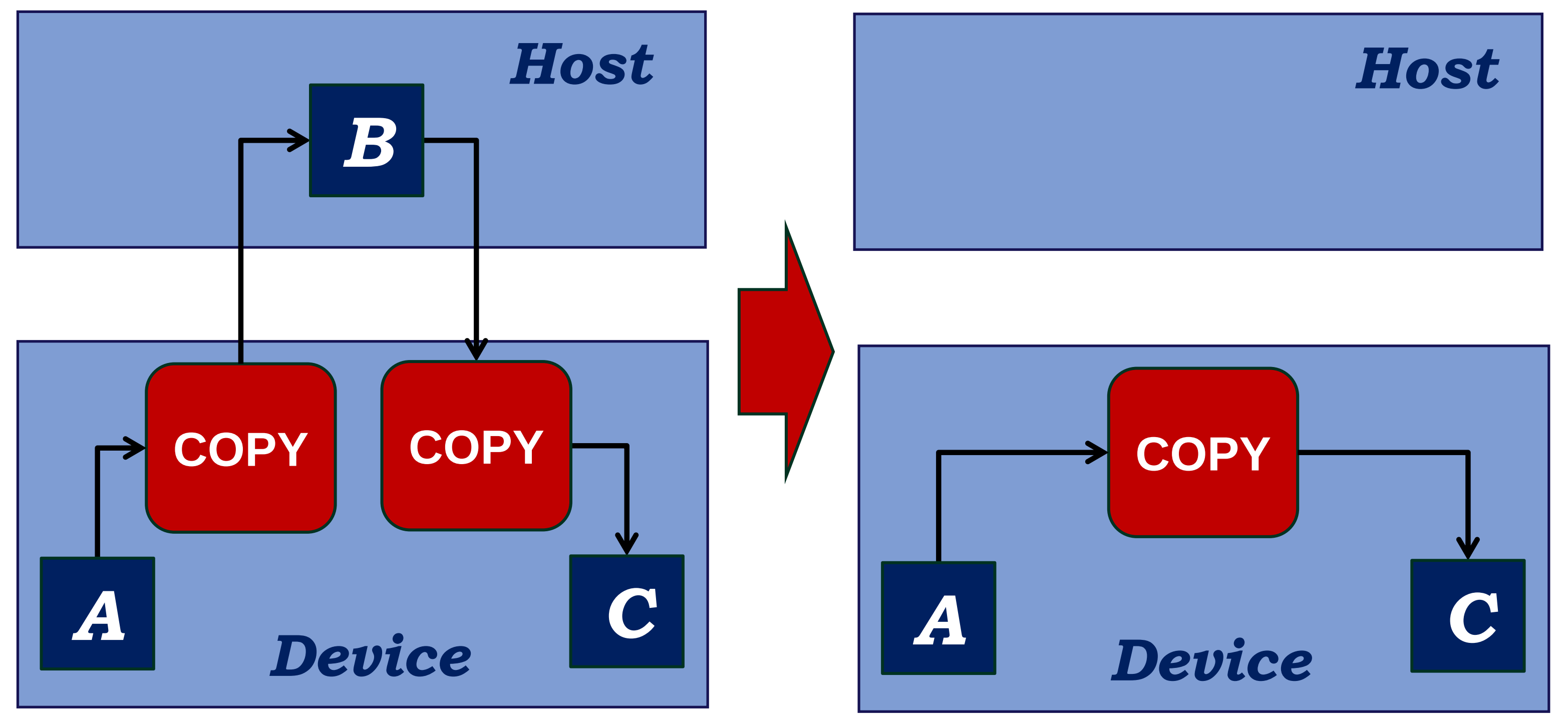

Figure 5. An example of one form of copy-elimination performed by Sequoia

Conclusion and Future Work

The Sequoia programming model makes it easy to write portable programs that can be mapped to GPUs. Sequoia enables optimizations that minimize the overhead of writing efficient GPU code. In addition, Sequoia gives the programmer the power to target multi-GPU machines.

No performance results are presented here as we are hoping to publish them in an upcoming paper. In addition to this there are several additional optimizations that we hope to complete before demonstrating Sequoia's performance on GPU accelerated machines.

Our planned future optimizations include:

• **Software Pipelining** – the ability to overlap communication with computation is an important scheduling optimization for the Sequoia compiler. We plan to implement this feature at two granularities: for kernels running on the device as well as for tasks running within CTAs.

• **Memory Coalescing** – we plan on using the compiler's knowledge of the program to optimize data layout in order to exploit the GPU's ability to do memory coalescing operations. This is even more important in the context of ECC which adds significant overhead to memory writes for supercomputing applications.

• **Irregular parallelism** – support for irregular parallelism will become an important feature as a wider variety of applications are beginning to be ported to GPUs