SPAP: A Programming Language for Heterogeneous Many-Core Systems

Qiming Hou[*] Kun Zhou[†] Baining Guo[*‡]

[*]Tsinghua University [†]Zhejiang University [‡]Microsoft Research Asia

Abstract

We present SPAP (Same Program for All Processors), a container-based programming language for heterogeneous many-core systems. SPAP abstracts away processor-specific concurrency and performance concerns using containers. Each SPAP container is a high level primitive with an STL-like interface. The programmer-visible behavior of the container is consistent with its sequential counterpart, which enables a programming style similar to traditional sequential programming and greatly simplifies heterogenous programming. By providing optimized processor-specific implementations for each container, the SPAP system is able to make programs efficiently run on individual processors. Moreover, it is able to utilize all available processors to achieve increased performance by automatically distributing computations among different processors through an inter-processor parallelization scheme. We have implemented a SPAP compiler and a runtime for x86 CPUs and CUDA GPUs. Using SPAP, we demonstrate efficient performance for moderately complicated applications like HTML lexing and JPEG encoding on a variety of platform configurations.

CR Categories: D.3.3 [Programming Languages]: Concurrent Programming Models—Language Constructs and Features

Keywords: programming model, heterogeneous platforms, programable graphics hardware

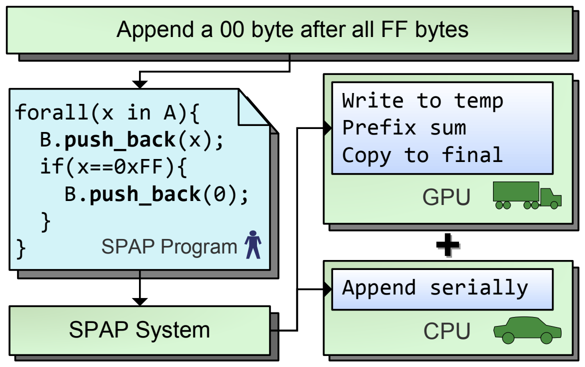

Figure 1: *The SPAP system architecture. The programmer writes a high level program using SPAP containers. The SPAP runtime automatically parallelizes the program to a heterogenous architecture using a variety of parallelization techniques.*

1 Introduction

Heterogeneous many-core architectures are increasingly used in client computing systems. Nowaday commodity systems, such as desktop computers and notebooks, are frequently shipped with one multi-core CPU (central processing unit) optimized for scalar processing and one many-core GPU (graphics processing unit) capable of general-purpose throughput processing. Application performance can be improved by orders of magnitude if such heterogeneous processing power is fully exploited by programmers.

An ideal programming language for heterogeneous systems should be architecture-independent. It should allow a programmer to write the *same program for all processors*, and the program should be able to not only perform efficiently on each individual processor but also utilize all available processors to achieve maximum performance. Realizing this ideal, however, is challenging due to the discrepancy among existing multi-core and many-core processing models. Processors with different processing models or even different processor vendors often have contradictory performance models spanning from instruction level to algorithm level. For example, on multi-core x86 CPUs it is beneficial to adjust the number of threads to the number of cores to avoid context switching costs, while on NVIDIA Geforce GPUs programmers are encouraged to maximize the number of threads to utilize the hardware latency-hiding scheduler. Such contradictory behaviors frequently motivate different algorithm choices on different processors.

Modern GPU programming languages like CUDA [NVIDIA 2009a], OpenCL [Khronos OpenCL Working Group 2008] and BSGP [Hou et al. 2008] are evolving to support general-purpose heterogeneous programming. OpenCL is designed to allow programmers to write kernel functions that may be compiled to both CPU and GPU, and similar efforts have been made for CUDA [Stratton et al. 2008]. However, in order to achieve efficient performance, programmers still have to write separate kernels for each processor because different processors may need different algorithms due to the processing model discrepancy. Consider prefix sum as an example. An optimized implementation for Geforce GPUs has to create a sufficient amount of threads and use a multi-pass parallel algorithm whereas on x86 CPUs a sequential sweep is usually more efficient. For a program to run efficiently on both GPU and CPU, the programmer has to implement both algorithms despite that either algorithm can run on both processors. Merge [Linderman et al. 2008] is a notable parallel programming framework for heterogeneous multi-core systems. It handles the processing model discrepancy using a predicate-based library system. Using Merge, a programmer can express computations using architecture-independent, high-level language extensions in the map-reduce pattern. The Merge system automatically selects the best available function implementations from the library for a given platform configuration. The system, however, still requires the programmer to provide optimized variants of each function for different processors to achieve high performance. As far as we know, most existing programming frameworks require programmers to write *different programs for different processors* to effectively utilize all available processors in a heterogeneous system.

In this paper, we propose SPAP (Same Program for All Processors), a container-based parallel programming language for heterogeneous many-core systems. The language provides a set of SPAP containers, each of which is a high level primitive with an STL (Standard Template Library)-like interface. An important property of SPAP containers is the *behavior consistency*, i.e., the programmer-visible behavior of a SPAP container is consistent with its sequential counterpart. For exmaple, in the program fragment in Fig. 1, `A` and `B` are two SPAP containers analogous to the STL `vector`. The programmer-visible behavior of the `B.push_back` operation is consistent with a serial STL `vector push_back`. In other words, the content of `B` after the `forall` loop enclosing SPAP `push_back` calls is exactly same as the content of an STL `vector` after a serial `for` loop enclosing STL `push_back` calls with similar arguments. Behavior consistency enables a programming style similar to traditional sequential programming, and thus greatly simplifies heterogeneous programming. Moreover, just like the wide use of STL in sequential programming, programmers are able to build complicated applications using only a few key SPAP containers such as resizable list, reduction and prefix sum. By providing optimized processor-specific implementations for each key container, the SPAP system is able to make SPAP programs efficiently run on individual processors. In short, SPAP containers effectively hide the processing model discrepancy with a combination of behavior consistency and optimized implementations.

SPAP also allows programmers to utilize all available processors of a heterogenous system to get increased performance. This is achieved by automatically distributing computations among different processors through an inter-processor task parallelization scheme. Programmers express computation tasks as a number of work units. The SPAP runtime system dynamically partitions the work units into subsets and dispatches them based on the availability and capacity of processors. The task partitioning and dispatching are performed iteratively until all work units are processed.

To summarize, this paper discusses the design and implementation of SPAP, a new programming language for heterogeneous many-core systems. Specifically, we make the following contributions:

- We propose SPAP, a container-based parallel programming language that allows the same program to work efficiently on all processors of a heterogeneous system and fully utilize the heterogeneous processing power.
- We implement a SPAP system, including a SPAP compiler and a runtime, for x86 CPUs and CUDA capable GPUs.
- We implement a variety of applications in SPAP, including an AES cipher, a HTML lexical analyzer and a JPEG encoder. For the JPEG encoder, heterogeneous processing is observed to deliver a 7.6× speed up on a quad-core CPU and a GPU relative to a well-optimized C implementation on a single-core CPU.

In the rest of the paper, we first describe the programming model of SPAP using source code examples. In Section 3, we detail the SPAP language constructs, followed by the description of the SPAP implementation for x86 CPUs and CUDA GPUs in Section 4. Section 5 evaluates our programming language using several examples. Section 6 reviews related work and Section 7 concludes the paper.

2 Programming Model

In this section we illustrate the programming model of SPAP from the programmer's perspective by using source code examples. The language syntax of SPAP is similar to BSGP [Hou et al. 2008], which in turn resembles C.

2.1 Containers

Consider a minor subproblem in JPEG encoding. Given a list of bytes `A` as input, insert a `0x00` padding byte after each `0xFF` byte in the list to form a new list `B`.

Listing 1 is the SPAP program for this task. The `forall` statement is the fundamental parallel construct in SPAP. A `forall` loop indicates that each iteration of the loop is completely independent except for SPAP container operations. All operations inside `forall`, including container operations, are completed once the control flow is returned to the code following the `forall` loop.

Listing 1 Padding byte insertion in SPAP

```
typedef unsigned char byte;
byte<> addPadding(byte<> A){
    auto B=new byte<>;
    forall(x in A){
        B.push_back(x);
        if(x==(byte)0xff){
            B.push_back((byte)0x00);
        }
    }
    return B;
}
```

Type `byte<>` declares a resizable list of bytes. Resizable list is a fundamental container in SPAP. The `push_back` operation appends elements to a list. It guarantees that once the enclosing `forall` loop completes, all elements will be appended to the list as if the `forall` loop is a sequential `for`/`foreach` loop.

Listing 2 x86 padding byte insertion in C++

```
vector<byte> addPaddingCPU(const vector<byte>& A){
    vector<byte> B;
    for(int i=0;i<A.size();i++){
        byte x=A[i];
        B.push_back(x);
        if(x==(byte)0xff){
            B.push_back((byte)0x00);
        }
    }
    return B;
}
```

Listing 3 Geforce padding byte insertion in BSGP

```
dlist(byte) addPaddingGPU(dlist(byte) A){
    B=new dlist(byte);
    int ntotal;
    spawn(A.n){
        //use scan to compute final offsets
        x=A[thread.rank];
        offset=(x==(byte)0xff?2:1);
        ntotal=scan(rop_add,offset);
        require{
            B.resize(ntotal);
        }
        //write the bytes to the computed offsets
        x=A[thread.rank];
        B[offset]=x;
        if(x==(byte)0xff){B[offset+1]=(byte)0x00;}
    }
    return B;
}
```

Listing 2 and Listing 3 are the C++ and BSGP code for the same task written for x86 CPUs and Geforce GPUs respectively. The x86 version serially appends the bytes to a standard C++ vector. Multi-core parallelization is not used due to the parallelization overhead and bus contention concerns. The Geforce version creates one thread for each input byte, computes its expected offset in the output list using a collective prefix sum (the `scan` function) and writes input/padding bytes to the output list in parallel. This algorithm is chosen to create sufficiently many threads to achieve maximum processor occupancy and thus maximize the effective memory bandwidth.

Note the algorithmic difference between Listing 2 and Listing 3. The programmer has to write and maintain both versions to achieve portability and efficiency. If OpenCL is used, one may compile either of the two algorithms to both processors. However, running Listing 3 on an x86 CPU would introduce considerable overhead from the collective scan while running Listing 2 on a Geforce GPU would result in degenerate performance due to the inability to utilize hardware latency hiding.

Listing 4 Main loop of a parallel 128-bit AES-CTR cipher

```
void encrypt(void* pdata,int sz,void* pkey,void* pctr){
    //Initialization
    int n=(sz>>4);
    uint rk[44];
    initRoundKey(rk,*(uint4*)pkey);
    auto l_FSb=new byte<256>;
    auto l_FT0=new uint<256>;
    memcpy(&l_FSb[0],FSb,sizeof(FSb));
    memcpy(&l_FT0[0],FT0,sizeof(FT0));
    uint3 c0=*(uint3*)pctr;
    //Partition block 0..n-1 across processors
    distribute(p0:p1 in 0:n-1){
        int m=p1-p0+1;
        auto p=new uint4<>;
        int base=p.mount((uint4*)pdata+p0,m);
        //p[base] now refers to ((uint4*)pdata)[p0]
        forall(i=0:m-1){
            uint4 x=make_uint4(c0.x,c0.y,c0.z,
                bigEndian((uint)(i+p0)));
            aesEncodeBlock(x,rk,l_FSb,l_FT0);
            p[base+i]^=x;
        }
        p.unmount();
    }
}
```

Using SPAP containers, the programmer only needs to write a single program as in Listing 1. At run time, the SPAP system detects available processors and substitutes respective optimized implementations for container operations. For x86 processors, the system replaces the SPAP `push_back` with an STL-like `push_back` when `forall` is executed on a single core. If `forall` is parallelized over multiple cores, the appended elements are redirected to per-core temporary lists that are merged at the end of `forall`. For Geforce GPUs, a temporary work space is allocated before `forall`, and `push_back` is replaced by writes to the work space. At the end of `forall`, the offset in the final output for each appended element is computed using a parallel prefix sum. Finally, the elements are moved from the work space to their respective final positions. Please refer to Appendix A for more details about the Geforce implementation.

2.2 Distributing Computations Among Processors

Now we demonstrate how to distribute computation across heterogenous processors using SPAP. Consider a 128-bit AES-CTR cipher [Federal ; Dworkin 2001]. The cipher splits a plain text into 128-bit blocks. Each block is assigned with a counter. The counters are AES encrypted using an input key and each text block is XOR (exclusive-or)-ed with its assigned encrypted counter to yield the cipher text. Since counters for all blocks are independent, all text blocks can be encrypted in parallel.

Listing 4 is the main loop of a heterogenous parallel AES-CTR cipher. During initialization, two AES lookup tables are copied to two SPAP lists for later use. Note that SPAP allows a native pointer to be obtained from a list using `operator[]` and `operator&`. The `distribute` statement is then used to partition computations into subsets and dispatch them to available processors. Each subset is dispatched to a processor, either a CPU core or a GPU with a dedicated CPU core that handles the corresponding GPU driver calls. Each processor then mounts a SPAP list `p` to its portion of the input data and uses `forall` to process `p`, utilizing in-processor data parallelism if available.

In the `distribute` statement, a global parallel task is partitioned into smaller subsets and dispatched to individual processors. The global task is abstractly represented as an integer interval `a:b` where every integer between `a` and `b` inclusively represents one work unit.

In Listing 4, one work unit corresponds to one plain text block. The `n` text blocks to be processed are represented as the integer interval `0:n-1`. Whenever a processor becomes available, a subset is split from the remaining task and dispatched to the processor. The subset size is determined by an integer measure of the processor's processing capability. For example, consider the case where a processor with capability `k` is available and the currently remaining portion of the global task is `a:b`. If `b-a>=k`, the task is split into two subsets `a:a+k-1` and `a+k:b`. `a:a+k-1` is dispatched to the processor and the remaining portion of global task is replaced by `a+k:b`. If `b-a<k`, task `a:b` is directly dispatched to the processor and the `distribute` statement exits after all processors have finished their subtasks. Fig. 2 illustrates an example task splitting and dispatching process.

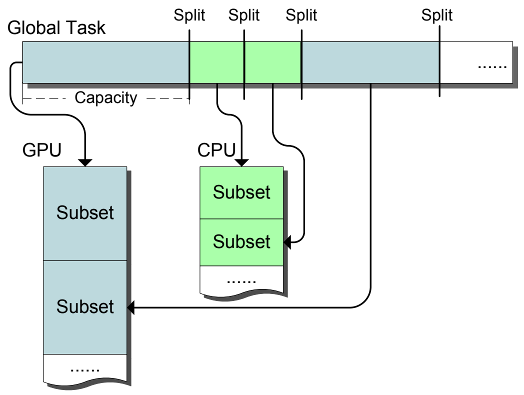

Figure 2: *Partition and dispatch a task to available processors.*

The capability of each processor should be chosen to be small enough to allow reasonable load balancing among all processors, and large enough to avoid introducing significant overhead on the processor. In SPAP, each processor has a default capability value optimized for work units consisting of a few tens or hundreds of arithmetic operations. When the default values are inappropriate, the programmer may specify alternative values.

2.3 Heterogeneous Processing with Containers

In this subsection, we use a more sophisticated example to demonstrate how to use SPAP containers in heterogeneous processing. Listing 5 is the code of a parallel prefix lexing [Hillis and Guy L. Steele 1986] pass in our parallel HTML lexical analyzer. This pass handles pointed brackets and quotes. The parallel prefix lexing algorithm computes the state of a lexing finite state machine at each character of an input string. It converts each character to a state transition table and computes a parallel prefix sum of the tables using a table composition operator. Our implementation further optimizes this algorithm by only computing the prefix sum at key characters, i.e., characters that correspond to non-identity state transitions.

In Listing 5, the work is first distributed to all available processors. A prefix sum container is constructed via `makePrefixSum`. The subsequent `forall` loops over all characters in the current subset to detect key characters. For each key character, its state transition table is added to the prefix sum container. Finally, a serialization task is created using the `serialize` construct to merge the results of all subsets.

The code block enclosed by `serialize` is converted to a sequential loop over all subsets and executed at the end of the enclosing `distribute` statement. For all subsets, the code block is executed

Listing 5 Parallel prefix lexing in HTML lexical analyzer

```
auto state=0; //Global initial state
auto allpos=new int<>; //Key charater positions
auto allst=new byte<>; //FSM states at key charaters
distribute(p0:p1 in 0:n-1){
    auto posi=new int<>;
    auto lexer=makePrefixSum(__portable__(byte a,byte b){
        //Table composition operator
        return
            ((b>>(((int)a<<1)&6))&(byte)3)+
            ((b>>(((int)a>>1)&6))&(byte)3)*(byte)4+
            ((b>>(((int)a>>3)&6))&(byte)3)*(byte)16+
            ((b>>(((int)a>>5)&6))&(byte)3)*(byte)64;
    },(byte)0xE4);
    //Loop over key characters
    forall"novector"(j=p0:p1){
        auto ch=(int)s[j];
        //Detect key chars: quotes / pointed brackets
        auto symid=((ch-1)<<2)&(8*3);
        int chstd=(0x273E3C22>>symid)&0xFF;
        if(ch==chstd){
            //Generate transition table
            auto tab=(byte)(0x6CE0E5D8>>symid);
            posi.push_back(j);
            lexer.push_back((byte)tab);
        }
    }
    byte end=lexer.total;
    //Merge subset results
    serialize{
        allpos.push_back(posi);
        //Compute final states from current global state
        forall(tab in lexer.values){
            int st=((int)tab>>(state*2))&3;
            allst.push_back((byte)st);
        }
        //Advance the global state to next subset
        state=((int)end>>(state*2))&3;
    }
}
```

in the creation order of the subsets, i.e., they are executed as if the `distribute` statement is a sequential loop. This is analogous to behavior consistency.

The `makePrefixSum` function creates a prefix sum container from an associative operator and a zero element. At the end of any `forall` loop that encloses `push_back` calls of the container, it returns the exclusive prefix sum of all appended elements as its `.values` member and the total sum as its `.total` member.

The string `"novector"` following the `forall` keyword is a hint to control the processor-specific code generation in the SPAP runtime. `"novector"` prevents the `forall` from being vectorized. In this example, the programmer found that vectorization does not significantly improve performance and added this hint to avoid generating unnecessary code.

2.4 Programming Model Summary

To summarize, SPAP supports two levels of parallelism – in-processor data parallelism through `forall`, and inter-processor task parallelism through `distribute`. This design is chosen to combine the strengths of the two levels.

The `forall` loop with container operations is the most fundamental programming pattern in SPAP since it allows an intuitive and optimization-friendly definition of SPAP container behaviors. From the programmer's perspective, `forall` resembles `for`, a common construct in sequential programming. Intuitively the behavior of `forall` is similar to `for`. This is used as a principle to guide our container behavior designs. On the other hand, we only guarantee container behaviors at `forall` completion points. Container operations that would otherwise require synchronization like `push_back` may be performed en masse as a postprocess. This allows container operations to be transparently mapped to optimized multi-pass algorithms on GPUs where the synchronization model is either weak or has high overhead.

While `forall` iterations may be directly partitioned across heterogeneous processors, such partitioning would be ignorant to data locality. Potentially expensive copies would have to be introduced implicitly to guarantee container behaviors. Due to the flexibility of container behaviors, it is difficult, if not impossible, to avoid or even predict such copies. Therefore, `distribute` is introduced to provide locality-conscious computation partitioning. Within each subtask generated by `distribute`, all `forall` loops are guaranteed to run on the same processor. Therefore, intermediate data produced and consumed within the same individual subtask will not cause implicit copies. This allows programmers to only reason about data locality issues when considering the input and output data of each `distribute`. Finally, to provide an analogy of behavior consistency, the `serialize` construct is provided to give programmers a way to merge subtask results with minimum concurrency reasoning.

Our memory model, i.e., the resizable list, is designed to be memory space oblivious and closely resembles DSM (Distributed Shared Memory). Lists may be randomly accessed on any processor without regarding where the data is actually stored. List data is implicitly copied if accesses to a list are performed on multiple processors. Like DSM, this semantic hides the underlying memory space from programmers.

3 Language Constructs

3.1 Forall

As introduced in Section 2.1, `forall` is the fundamental parallel construct in SPAP. At run time, the code inside each `forall` loop is parallelized and compiled ahead of time to native code on each available processor architecture. Currently, the following parallelization techniques are supported:

- Fine grained data parallel. One thread is created for each loop iteration. This technique is designed for many-core architectures such as a GPU.
- Coarse grained task parallel. The entire loop range is split into a global queue of equal-sized chunks. Processor cores fetch and process chunks from the queue in parallel. This technique is designed for multi-core CPUs.
- No parallelization. The `forall` loop is executed as a sequential loop. This technique is a fallback in case the available parallelism cannot overcome the parallelization overhead.
- Vectorization. The loop is vectorized using processor-specific SIMD instructions. Vectorization may be used jointly with any of the above techniques if the corresponding processor has vector instructions.

Note that it is possible for multiple parallelization techniques to be applicable on the same platform. In that case, the SPAP runtime uses a dynamic self-configuring system to choose a competent variant after a few timed executions. For details, please refer to Section 4.4. Alternatively, the programmer may specify parallelization preferences using hints.

In a `forall` loop, external variables may be read but cannot be written. The runtime copies accessed external variables to the appropriate memory spaces of available processors. Also note that the iterations of a forall loop are not allowed to synchronize or communicate with each other.

3.2 Resizable List

Resizable list is the fundamental container in SPAP. It is also the only guaranteed portable way of accessing memory. A resizable list supports three operations in forall loops:

- operator[] indexes an element in the list. It may be used to read/write arbitrary list elements. operator[] follows the acquire/release consistency with the forall entry/exit as the acquire/release points.
- push_back appends an element into the list. As introduced in Section 2.1, when the enclosing forall ends, the elements are appended to the list as if the forall were a sequential loop.
- add also appends an element into the list. When the enclosing forall ends, all elements are appended to the list exactly once but in undefined order.

The three operations are mutually exclusive in forall loops. For each list in each forall, only one of the three operations can be used. Outside forall loops, the three operations are also supported except they are no longer mutually exclusive and add is equivalent to push_back. Common container operations like new, delete, resize and reserve are also supported. None of the list operations are thread-safe outside forall loops, and a per-list lock is provided as two methods lock and unlock.

The resizable list implementation is provided by the SPAP runtime. For details, please refer to Section 4.3.

3.3 Distribute

The distribute construct splits a task into subsets and dispatches them to individual processors. Within distribute, forall loops appear to be atomic. forall writing the same list in different subsets are implicitly serialized using locks. List accesses outside forall loops are not atomic. The programmer is responsible for serializing them using lock and unlock methods of the lists.

We also provide atomic sections in distribute to help programmers to deal with concurrency related problems. Atomic sections are code blocks enclosed in atomic{} and are executed atomically. Currently we implement atomic sections using a system-wide lock.

3.4 Miscellaneous

Native Code Interface Our language allows SPAP code and native code to be mixed in the same file. As illustrated in the code examples, forall loops are directly inserted into native code and SPAP resizable lists are manipulated as native objects. We also provide a function annotation, __portable__, to distinguish SPAP functions from native functions. __portable__ functions may be called from both SPAP code and native code, but cannot call native functions except in processor-specific sections (described later in this section). For CUDA/BSGP compatibility, we also provide a __device__ annotation to indicate SPAP functions that can only be called from SPAP code.

We also provide two methods, mount and map, to allow data exchange between SPAP resizable lists and native pointers. mount binds a list to a native pointer and map obtains a native pointer to a range of list elements. Native pointers may also be obtained from lists by using operator& with operator[]. For a code example of mount, please refer to Listing 4. Note that mount may fail if the input pointer does not satisfy the alignment requirement of the list implementation. In that case, a.mount(...) returns a base subscript base so that a[base] refers to the element at the input pointer.

Processor-Specific Section An if(targeting("xxx")) statement is provided to test the targeting platform and insert a section of platform-specific code. It is useful for low level optimization on specific processors.

Listing 6 Portable optimized function for float to 8-bit integer conversion

```
__portable__ int fast8bit(float f){
    if(targeting("CUDA")){
        //CUDA GPUs have a dedicated instruction
        return __float2int_rn(f);
    }else if(targeting("x86")){
        //On x86, exploiting IEEE754 format is faster
        return __float_as_int(f+8388736.f)^0x4b000080;
    }else{
        //Revert to portable code on other processors
        return (int)floor(f+0.5f);
    }
}
```

Listing 6 is an optimized function to convert a floating point number to its nearest integer. By utilizing the if(targeting("xxx")) statement, the function compiles to respective optimized implementations on CUDA enabled GPUs (like GeForce) and x86 CPUs while it reverts to a portable version on other processors.

Hinting Optional hints may be supplied at forall statements for manual parallelization control. Hints are written as a string literal following the forall keyword as illustrated in Listing 5.

Standard Containers The runtime provides a library of standard SPAP containers for which an efficient portable implementation is difficult or impossible. The following is a list of the standard containers supported in our current SPAP system:

- CPersistentVariable<typename T> defines a variable of type T that is persistent across iterations in the enclosing forall loop when the loop is executed sequentially. If the forall loop is not executed sequentially, CPersistentVariable behaves as an ordinary variable which is reset to a programmer-specified initial value at the beginning of each iteration.
- makeTotal(op, z) creates a reduction container for a commutative associative operator op whose zero element is z.
- makePrefixSum(op, z) creates a prefix sum container for an associative operator op whose zero element is z.
- CHistogram<int N> creates a histogram container that computes a histogram for integers between 0 and $N - 1$ inclusively.

We plan to add containers for sorting, irregular reduction and disk I/O in the near future.

4 Implementation

4.1 General Pipeline

Fig. 3 illustrates the pipeline of our SPAP system. Currently the system consists of a bytecode compiler, a parallelizing runtime compiler and a runtime library. forall loops are first compiled

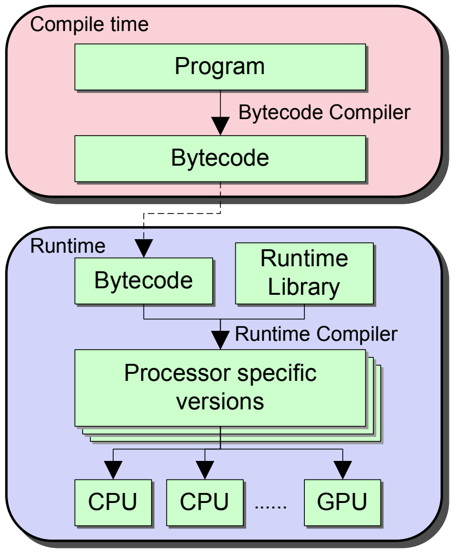

Figure 3: *The SPAP system pipeline.*

to bytecode fragments. At run time, the bytecode fragments are parallelized and compiled to available processors by the runtime compiler.

In order to support processor-specific sections, all operations, including arithmetic operations of basic types, are represented using function calls in our bytecode. For each function, a unique string is stored in the bytecode to store its name, parameter list and processor type. The runtime compiler uses this information to convert function calls in the bytecode to its IR (Intermediate Representation) instructions or calls to runtime library functions.

4.2 Standard Containers

The standard containers are implemented using a combination of code reordering constructs, processor-specific sections and hard-coded compiler-based translations. List and `CPersistentVariable` work as a basis for implementing other containers. Their operations directly map to bytecode operations and are translated by the runtime compiler. For higher level containers, we borrow and generalize the BSGP `require` [Hou et al. 2008] construct to provide a way to interact with the runtime compiler from high level source code. The runtime compiler defines a number of significant code locations for parallelization techniques. In container implementations, `require` is used to insert platform-specific code into these significant locations on a per-container basis. Each `require` statement takes a string for the location name and a block of code to be inserted. For example, one may write `require("x86.init"){a=new int<>;}` to create a list `a` during the initialization of the x86 version. Using `require`, lists, `CPersistentVariable` and processor-specific sections, we are able to implement all other containers with moderate difficulty.

4.3 Resizable List

An important challenge in implementing the resizable list system is to allow a list to be randomly accessed from both CPUs and GPUs. In CUDA, the simplest way to achieve this is to use its "mapped host memory", i.e., mapping CPU memory into GPU address space. However, this approach has three problems:

- Expensive PCI-Express bus data transfers are incurred every time the memory is accessed from GPU. CUDA does not provide any built-in caching mechanism.
- Mapped host memory is page locked and cannot be swapped out by the CPU-side OS. It makes the entire system slow and unstable when allocated in large quantities.
- Not all CUDA enabled GPUs support mapped host memory.

To avoid these issues, we implement lists using VM (virtual memory) based techniques analogous to software distributed shared memory [Roy and Chaudhary 1998]. A replica of each list is maintained on both CPU and GPU. Consistency between the replicas is maintained by invalidating pages written on the other processor. When invalidated pages are accessed, the actual content is copied from the replica on the other processor in a page fault handler. Since currently CUDA GPUs do not have programmable VM subsystems, special care needs to be taken to avoid GPU-side VM operations. We avoid invalidating GPU pages by eagerly synchronizing CPU updates to GPU. Pages modified by GPU are detected using compile-time access pattern analysis. Currently, the access pattern analysis only recognizes "coalesced" access patterns, i.e., writes with subscripts in the form of the `forall` loop variable plus a loop invariant value. When there are unrecognized access patterns, the entire CPU replica is invalidated.

4.4 Parallelization and Variant Selection

Parallelization of the `distribute` level is handled entirely by the compiler frontend. The code block enclosed in each `distribute` is converted to a function object and the `distribute` is converted to a call that invokes a heterogeneous scheduler with the function object as a parameter. Parallelization of the `forall` level is done by the runtime as described in Section 4.1. Currently for each `forall` a maximum of three versions may be generated - sequential x86, vectorized x86 and data parallel CUDA. `forall` loops outside `distribute` may also be parallelized across multiple CPU cores. Such multi-core parallelization is done by splitting the `forall` loop range and invoking the sequential or vectorized x86 version on the subranges on individual cores in parallel.

When multiple parallelization approaches are applicable for a given `forall`, the runtime system has to make decisions and choose a competent approach. In addition, for `forall` loops outside `distribute`, the subrange size into which the multi-core approach splits the loop range needs to be tuned. We developed a dynamic self-configuring system to make these decisions and tune the subrange size. Currently the system makes three decisions in the following order: CPU versus GPU, sequential versus vectorized, and single-core versus multi-core. Note that if the first decision is the GPU parallelization approach, there is no need to make the other two decisions. The single-core versus multi-core decision is made after the more efficient per core approach is found during the sequential versus vectorized decision. The decision results are permanent. Once a decision is made, its result is saved to disk. After all decisions are made, no more experiments need to be done and the chosen technique is used in all subsequent executions.

Sequential versus vectorized and single-core versus multi-core decisions are made via pairwise comparisons. During the first few executions of each `forall`, the system executes two timed test runs of two equal-sized subranges of the `forall` loop range using two candidate parallelization techniques. After doing a fixed number of comparisons, the candidate that wins in more tests is chosen as the final technique. The remaining portion of the loop range is executed using this final technique. A number of optimizations are made to improve the stability and minimize the overhead of the decision making process. Please refer to Appendix B for more details.

The subrange size for parallelizing multi-core `forall` is iteratively tuned to make the processing time for each subrange above an empirical threshold T_0. At the end of each `forall`, the subrange size s is updated to $s' = \max\left\{s, \frac{T_0}{T} n\right\}$, where T is the `forall` execution time and n is the number of iterations. T_0 is empirically chosen to be large enough to prevent the multi-core scheduler from introducing significant overhead while small enough to yield satisfactory load balance.

The CPU versus GPU decision is more complicated than purely CPU-side decisions as it depends on the problem scale. GPU may be more efficient than CPU when there are a sufficiently large number of iterations in the `forall` loop, while CPU is always more efficient when the processing cost of the entire `forall` is less than the GPU kernel launch overhead. Our solution is to find a proper threshold – the GPU approach is used when the iteration count is above the threshold and the CPU approach is used otherwise. The threshold is determined using a binary search like method based on timing comparisons of CPU and GPU approaches. For more details about the threshold tuning, please refer to Appendix C. Note that the CPU versus GPU decision only needs to be made for `forall` loops outside `distribute`. In `distribute`, the CPU versus GPU decision is solely made according to the type of the available processor to avoid violating data locality assumptions.

5 Experimental Evaluation

In this section we use several examples to evaluate the performance of our SPAP system on x86 CPUs and CUDA GPUs. As mentioned, an important advantage of SPAP is that it greatly simplifies heterogeneous programming by providing portable high level containers. This assessment is necessarily subjective and the best way to verify it is to examine SPAP source code and compare the programming style with alternative programming environments. For this reason, we provide the SPAP source code of our JPEG encoder in Appendix D in addition to the code samples in Section 2.

Machine	CPU	GPU
1	Intel Xeon 3.73GHz ×2	8600GT (32 ALUs)
2	Intel Xeon 3.73GHz ×2	9800GT (112 ALUs)
3	AMD Phenom 2.60GHz ×4	GTX280 (240 ALUs)

Table 1: *Test machines used in this paper.*

Our evaluation focuses on two points - the overall potential of heterogeneous processing using SPAP and the quality of processor-specific code generated from behavior consistent containers. We implemented three examples from different application fields and tested them on a variety of architectures. Table 1 lists our test machines. The tested GPUs span all three existing generations of the NVIDIA GeForce brand. The three examples we implemented are:

- AES encrypts a file using the AES-CTR algorithm [Federal ; Dworkin 2001]. It is a simple, embarrassingly parallel workload that evaluates an arithmetic intensive function independently on many input blocks.
- HTML generates the list of tags and data contents from a HTML file. It is a moderately complicated workload that involves a few behavior consistent container operations like prefix sum and `push_back`.
- JPEG is a JPEG image encoder. It is a realistic application and involves a few processing steps with different parallelization characteristics.

Table 2 lists the raw performance data for all examples on all test

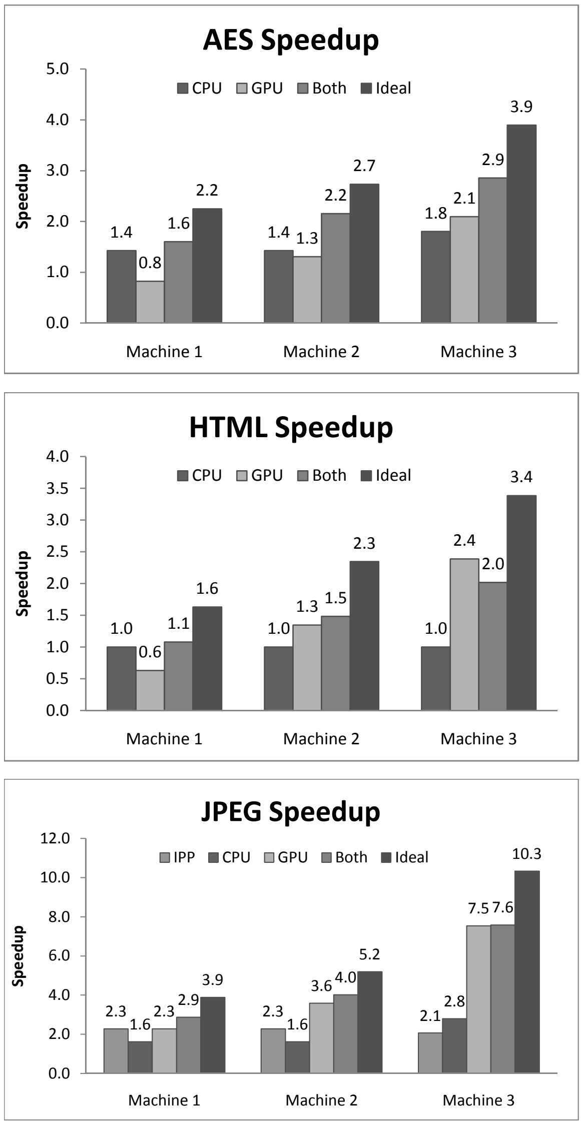

Figure 4: *Speedup factors comparing to baseline. For the JPEG example, Intel IPP speedup is also provided as a reference.*

machines. Note that for each example, we only need to write one SPAP program. For each machine, three versions of each example are tested by using hints to restrict the program to run on three configurations, one on CPU only, one on GPU only and one on both CPU and GPU. For each example, we also run a CPU baseline implementation to provide reference performance data. For AES and JPEG, the implementations in Crypto++ and libjpeg are used as baseline implementations. For HTML, we used the CPU restricted version of our SPAP program as the baseline since there are no publicly available implementations. Timings of the JPEG example include the time taken to write the output file due to the difficulty of separating output code from the processing code in libjpeg. I/O time is excluded in other examples. Fig. 4 shows the speedup relative to baseline implementations, and ideal heterogeneous speedups are shown as the "ideal" bars. The ideal heterogeneous speedup is computed by combining the CPU and GPU processing time assuming an ideally balanced workload, i.e., the harmonic mean of the CPU and GPU processing time.

The potential of heterogeneous processing has been clearly demon-

			Machine 1				Machine 2				Machine 3			
	Baseline	Input size	Base	CPU	GPU	Both	Base	CPU	GPU	Both	Base	CPU	GPU	Both
AES	Crypto++	121MiB	679	476	825	424	679	476	520	315	563	312	269	197
HTML	SPAP CPU	17MiB	190	190	301	176	190	190	141	128	105	105	44	52
JPEG	libjpeg	121MiB	2920	1810	1285	1018	2920	1810	815	727	2532	905	336	334

Table 2: *Raw performance measurement. All data represent processing time in milliseconds.*

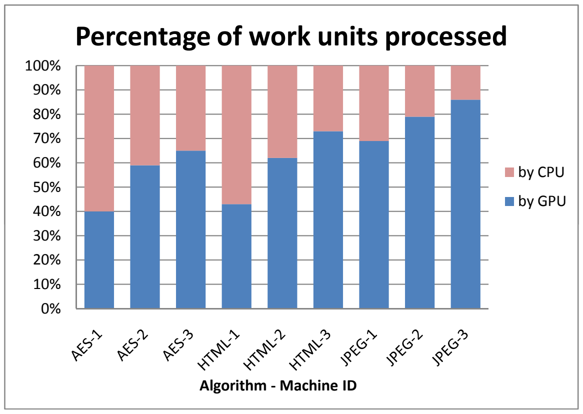

Figure 5: *The percentage of work units assigned to CPU and GPU.*

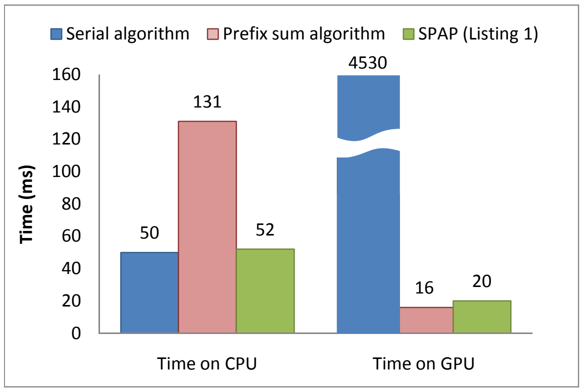

Figure 6: `push_back` *performance comparison between three implementations of the padding byte insertion problem in Section 2.1.*

strated. The heterogeneous version consistently achieves a notable speedup against the baseline. The results on Machine 1 show that heterogeneous programming allows the overall performance to benefit from the addition of a GPU even when a pure GPU version does not bring any acceleration. As a result, heterogeneous programming allows performance to be improved transparently by installing or upgrading GPUs, without risking potential performance degradations that pure GPU approaches may suffer from when the installed GPU is slower than CPU. On the other hand, our heterogeneous processing speedup still has not reached the ideal level. The heterogeneous version may even be slower than a pure GPU program when the GPU processing time is too short (e.g., HTML on Machine 3). This problem may be caused by an overhead introduced at both the CPU side and the GPU side when the CUDA CPU-GPU data transfer and memory intensive CPU processing are performed simultaneously. We suspect this is caused by the CPU-side bus contention between the CPU tasks and the internal code in the CUDA driver. For heterogeneous processing to be beneficial, the performance gain of CPU processing has to outweigh such overhead. Currently we are unable to work around this problem. Nevertheless, Fig. 4 shows that heterogeneous processing on CPU and GPU is able to outperform CPU (or GPU) alone in a majority of situations.

Fig. 5 lists the percentage of work units executed on CPU and GPU for all example-machine combinations. In general, more computations are distributed to GPU as the GPU becomes faster. GPU is capable of processing more work units in the floating point intensive JPEG example than the integer intensive AES and HTML examples. This result shows that the computation partitioning routine in our `distribute` construct adapts to different platform configurations reasonably well.

Fig. 6 compares the execution time of different algorithms of the padding byte insertion problem described in Section 2.1 on different processors. The serial algorithm in Listing 2 and the prefix sum based algorithm in Listing 3 are implemented on both CPU and GPU, and are compared with the corresponding CPU/GPU restricted versions of the SPAP program in Listing 1. The test machine used is Machine 3. The CPU implementation of the the prefix sum algorithm incurs approximately a 160% overhead. The GPU implementation of the serial algorithm results in degenerate performance as GPU is not optimized for scalar processing. The SPAP system is able to hide such processing model discrepancy and allows Listing 1 to achieve satisfactory performance on both processors. Note that the SPAP program is slightly less efficient than the prefix sum algorithm (Listing 3) on GPU. This is because our container interface design does not allow recomputing the appended elements like in Listing 3 and the elements have to be temporarily written to memory. Nevertheless, we are still able to achieve satisfactory performance.

We also evaluate the quality of code generated from SPAP containers by comparing application performance with highly-optimized processor-specific implementations. The JPEG example is selected as the basis of this comparison. First, we compare our CPU version of JPEG with the IPP (Intel Performance Primitives) library, a highly-optimized library supplied by Intel. We modified the timing code in the `ijg_timing.c` example in IPP 6.1 to print the JPEG encoding time in milliseconds. For the test image we used, IPP takes 1280ms on Machine 1/2 and 1228ms on Machine 3. Our CPU version performs competitively by taking 1810ms on Machine 1/2 and 905ms on Machine 3 respectively. On the GPU side, our GPU version achieves a 3.6× speed up over the libjpeg baseline on a GPU with 112 ALUs. This is competitive against the latest published results [Mou and Xing 2008; Wu et al. 2009] we are aware of, which reported 3.4× and 2.9× speed ups respectively on a GPU with 128 ALUs.

6 Related Work

Our SPAP language combines many elements from existing works. The `forall` semantic and DSM-like list are influenced by Chapel [Callahan et al. 2004] and ZPL [Chamberlain et al. 2000]. The `distribute` construct resembles the `mappar` construct in Sequoia [Fatahalian et al. 2006]. The idea of simultaneously processing on both CPU and GPU is inspired by Merge [Linderman et al. 2008], Harmony [Diamos and Yalamanchili 2008] and OpenCL [Khronos OpenCL Working Group 2008]. The resizable list operations are influenced by Direct3D buffers [Blythe 2006] and BSGP collective operations [Hou et al. 2008]. An important difference between our work and these previous works is the concept of behavior consistency. In SPAP, high-level behavior consistent containers are provided to hide concurrency and performance model discrepancies. This allows many problems to be implemented as unified programs that are able to work efficiently on heterogenous processors.

The Merge framework [Linderman et al. 2008] is also able to hide processing model discrepancy by providing a library of function variants. Although some SPAP container operations may be emulated using functions on certain architectures, it is very difficult, if not impossible, to completely implement SPAP containers using a function library. For example, on data parallel architectures like Geforce, many key container operations (e.g., `push_back`) have to be implemented using multi-pass algorithms which contain many separated steps. A few specific steps (e.g., temporary space management) have to be interleaved with the system-defined parallelization code that does not correspond to any container operation calls. The multi-pass algorithms cannot be mapped to simple functions which can only abstract processing at container operation calls.

Compared to concurrent containers [Intel], the SPAP container semantic is stronger with respect to programmer-visible behavior and weaker with respect to concurrency. SPAP containers guarantee consistent programmer-visible behaviors with their sequential counterparts, but such a guarantee only applies at `forall` boundaries. In contrast, concurrent containers only guarantee thread-safe behaviors while its guarantee holds everywhere in a program. Neither the SPAP container nor the concurrent container may replace each other.

Our container semantics resemble the reducer [Frigo et al. 2009] in Cilk++. The key difference is that SPAP containers are designed to fully utilize heterogeneous platforms whereas Cilk++ reducers are designed for a work stealing environment for multi-core CPUs. SPAP containers allow efficient implementation on data parallel GPUs where a work stealing environment is impractical to implement and/or significantly less efficient than hardware schedulers. In particular, we have demonstrated efficient SPAP container implementations on Geforce GPUs which do not support general function call stacks, a fundamental ingredient required by the reducer semantics definition.

Shared memory for heterogeneous processors has also been proposed in [Saha et al. 2009]. Our list system differs from their work in that our system may be implemented on existing more restrictive architectures like Geforce at the cost of not supporting pointers.

7 Conclusion

We have presented SPAP, a new container-based programming language for heterogenous many-core systems. SPAP abstracts away processing model specific concerns using high-level behavior consistent containers. It allows programmers to write unified programs that are able to run efficiently on heterogeneous processors.

The SPAP system is still in the early stage of development. In the future, we plan to add more containers to the standard library. To add a new container, we need to provide optimized implementations for all known processing models and parallelization techniques. This is a necessary tradeoff as our system abstracts processor/parallelization specific concerns in the container layer. Second, we want to exploit more general functionalities of upcoming GPU architectures like Larrabee [Seiler et al. 2008] and Fermi [NVIDIA 2009b] to broaden the range of SPAP container functionalities. It is also interesting to generalize the behavior consistency to more high-level parallel constructs like parallel recursion and nested parallelism in addition to our current parallel loops. Finally, we plan to port SPAP to more architectures like AMD Radeon and CPU/GPU clusters.

References

B , D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3, 724–734.

C , D., C , B. L., Z , H. P. 2004. The cascade high productivity language. *High-Level Programming Models and Supportive Environments, International Workshop on 0*, 52–60.

C , B. L., C , S.-E., L , E. C., L , C., S , L., W , W. D., M , S. 2000. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering 26*, 2000.

D , G. F., Y , S. 2008. Harmony: an execution model and runtime for heterogeneous many core systems. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, ACM, New York, NY, USA, 197–200.

D , M., 2001. NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation - Methods and Techniques.

F , K., K , T. J., H , M., E , M., H , D. R., L , L., P , J. Y., R , M., A , A., D , W. J., H - , P. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

F , A. T. Processing standards publication 197.

F , M., H , P., L , C. E., L -B , S. 2009. Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ACM, New York, NY, USA, 79–90.

H , W. D., G L. S , J. 1986. Data parallel algorithms. *Commun. ACM 29*, 12, 1170–1183.

H , Q., Z , K., G , B. 2008. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Gr. 27*, 3, 9.

I . Intel TBB (Thread Building Blocks) homepage. http://www.threadingbuildingblocks.org/.

K O CL W G , 2008. The OpenCL Specification, Version 1.0.

L , M. D., C , J. D., W , H., M , T. H. 2008. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not. 43*, 3, 287–296.

M , D., X , Z., 2008. A Simple JPEG Encoder With CUDA Technology.

NVIDIA, 2009. CUDA introduction page. http://www.nvidia.com/object/cuda_home.html.

NVIDIA, 2009. Fermi introduction page. http://www.nvidia.com/object/fermi_architecture.html.

R , S., C , V. 1998. Strings: A high-performance distributed shared memory for symmetrical multiprocessor clusters. In *in Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pp.

S , B., Z , X., C , H., G , Y., Y , S., R , M., F , J., Z , P., R , R., M , A. 2009. Programming model for a heterogeneous x86 platform. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ACM, New York, NY, USA, 431–440.

S , L., C , D., S , E., F , T., A , M., D , P., J , S., L , A., S , J., C , R., E , R., G , E., J , T., H , P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3, 1–15.

S , J., S , S., H , W. 2008. MCUDA: An efficient implementation of cuda kernels for multi-core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*.

W , L., S , M., C , D., 2009. CUDA WUDA SHUDA: CUDA Compression Project.

Appendix A: CUDA `push_back` Implementation

Our CUDA `push_back` implementation uses a multi-pass algorithm. The largest available continuous block of GPU memory is reserved as a global temporary list before the enclosing `forall` statement. During the `forall` loop, each thread independently writes appended elements to a private work space allocated from this global temporary list. At the end of each thread, the starting address of its private work space and the number of elements it has appended are saved. After the `forall` loop, a prefix sum is used to compute the final address in the result list for the elements appended by each thread. A final kernel is launched to copy elements from per-thread private work spaces to their respective final addresses in the result list.

The key component in this algorithm is the per-thread private work space allocation. This step has to be implementable on all existing GeForce GPUs, i.e., it has to be implemented without using any atomic operations. Our solution is to split the entire global work space into a fixed number of equal-sized pools and assign each logical thread to a pool based on the thread's physical SM (Streaming Multiprocessor) id and in-SM thread id. Such an assignment guarantees that no simultaneously executing threads will append to the same pool and completely eliminates the need of atomic operations. Each thread loads the tail pointer of its pool to a register at its beginning and stores it at its end. The allocation at each `push_back` simply increments the tail pointer.

Note that the algorithm fails if the size of elements appended to any pool exceeds the pool's size. Ideally, the number of elements appended to each pool should be balanced to minimize failures when sufficient memory is available. Our pool allocation strategy is based on the physical execution unit assignment. Pool utilization is automatically balanced as the GPU hardware thread scheduler balances thread workload.

We also optimized two special cases of `push_back`. When exactly one `push_back` is called per iteration for a given list, a `resize` is inserted before the `forall` and the `push_back` is converted to an ordinary store. When at most one `push_back` is called per iteration for a given list, the `push_back` is converted to a call to the BSGP `compact` collective primitive at the end of the `forall`.

Appendix B: Optimizations for Pairwise Comparisons between Parallelization Approaches

While the raw idea of comparing timings of two parallelization approaches to find the faster one is relatively simple, in practice many optimizations are required to minimize the impact of timing errors and reduce the overhead of timing the slower approach.

To make the comparison more reliable, a comparison result is discarded if the running time of either candidate is shorter than T_{sleep}. T_{sleep} is an approximation of the OS task switch interval currently measured as the time of a `Sleep(1)` OS call. We expect T_{sleep} to be significantly larger than a majority of low-level timing error sources like the cache miss, TLB miss and page fault while still small enough to remain unnoticeable to programmers.

Two optimizations are employed to minimize the overhead introduced by the slower test candidate. The first is to impose an upper bound on the `forall` subrange size used in comparisons. This makes sure that a majority of the loop range will be executed only by the winning candidate in the comparison. The upper bound is initially set to infinity. After each comparison, if the currently faster candidate takes more than $10T_{sleep}$ to process the current comparison subrange, the upper bound is reduced to half of the current subrange size. The second optimization is to allow early termination when one parallel approach is significantly more efficient than the other. After each comparison, if one candidate wins by more than $5T_{sleep}$, it is chosen as the final winner without further comparisons.

Appendix C: CPU-GPU Transition Threshold Tuning

As mentioned in Section 4.4, the threshold for selecting CPU/GPU parallelization approaches is determined via a binary search like method. At initialization, the threshold is first set to $768N_{SM}$ where N_{SM} is the number of multiprocessors in the GPU. This value is an empirical estimation of the required number of threads to fully utilize the parallelism on GPU. After every `forall` execution, the threshold is increased if the CPU approach is faster and decreased if the GPU approach is faster. The increase and decrease are performed by multiplying a constant factor. The threshold is fixed the first time the comparison result reverts, i.e., the first time the winner approach changes.

Special care is required for the CPU versus GPU timing comparison. For a given `forall`, there are two possibilities for the transition point. When CPU is consistently faster than the GPU for all loop range sizes, the transition point is at positive infinity. In our experience, this case rarely occurs and we currently do not handle it. When GPU is faster than CPU for large loop ranges, the point CPU processing time exceeds GPU launch overhead may be used as a reasonably accurate transition point. In this case, the timing results during threshold tuning may be highly noisy as the GPU launch overhead is comparable to timing errors like the OS task switch time. We developed two mechanisms to alleviate this problem. The first mechanism is to filter noises by taking the most common outcome of multiple comparisons. The threshold is only increased or decreased if a number of continuous measurements yield the same result. The second mechanism is to approximate the GPU launch overhead as the minimal execution time of all timed GPU executions. Since all system errors in execution time measurements are positive, the minimal value typically becomes stable after a small

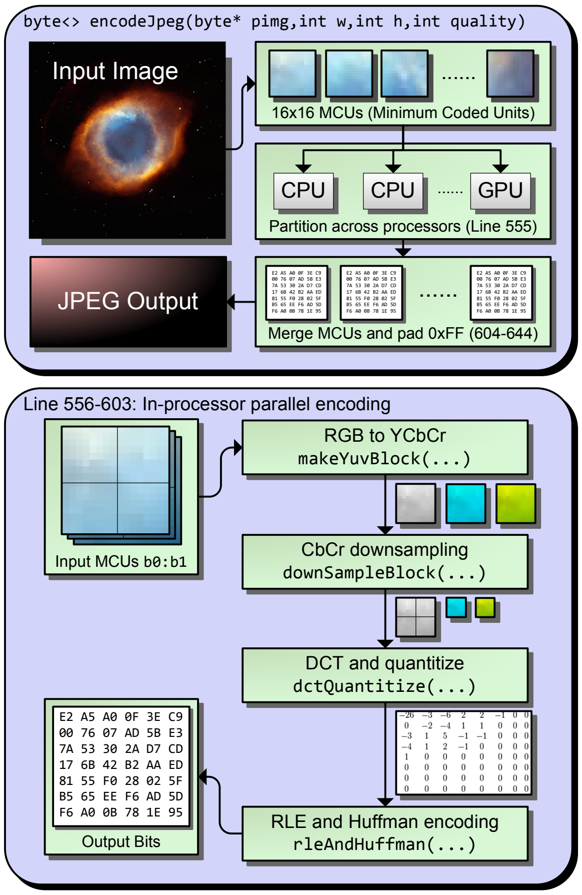

Figure 7: *Flowchart of the JPEG Encoder*

number of timed GPU executions. The minimum approximation may be expected to be reasonably accurate since when the available parallelism are not fully utilized on existing GPU architectures, the execution time is dominated by the kernel launch overhead and the sequential execution time of one `forall` iteration.

Appendix D: JPEG Encoder Source Code

```
/*
non-bottleneck code, type and tables are copied from
  Cristian Cuturicu's 1999 simple jpeg encoder
specialized to little endian architecture
*/
#include <windows>
#include <emmintrin>
#include "jpeg_type_table.h"

typedef unsigned char byte;
typedef unsigned int uint;

inline int wordSwap(int a){
   a&=0xffff;
   return ((a>>8)|(a<<8))&0xffff;
}

// Set quantization table and zigzag reorder it
void set_quant(BYTE *basic, BYTE quality,
      BYTE *newtable){
   int i;
   long temp;
   for (i=0; i<64; i++){
      temp=((long)basic[i]*(long)quality+50L)/100L;
      // limit the values to the valid range
      if(temp<=0L)temp=1L;
      if(temp>255L)temp=255L;
      newtable[zigzag[i]]=(BYTE)temp;
   }
}

static float t_Y[64];
static float t_Cb[64];
void prepare_quant_tables(){
   static double a[8] = {1.0, 1.387039845, 1.30656296,
   1.17587560, 1, 0.78569496, 0.5411961, 0.275899379};
   BYTE row, col;
   BYTE i=(byte)0;
   for (int row=0;row<8;row++){
      for (int col=0;col<8;col++){
         t_Y[i]=(float)(1.0/(
            (double)DQTinfo.Ytable[zigzag[i]]*
            a[row]*a[col]*8.0));
         t_Cb[i]=(float)(1.0/(
            (double)DQTinfo.Cbtable[zigzag[i]]*
            a[row]*a[col]*8.0));
         i++;
      }
   }
}

void initDQT(BYTE q){
   DQTinfo.marker = wordSwap(0xFFDB);
   DQTinfo.length = wordSwap(132);
   DQTinfo.QTYinfo = 0;
   DQTinfo.QTCbinfo = 1;
   set_quant(std_luminance_qt,q,DQTinfo.Ytable);
   set_quant(std_chrominance_qt,q,DQTinfo.Cbtable);
   prepare_quant_tables();
}

__portable__ float fastfloatu(int c){
   if(targeting("CUDA")){
      return (float)c;
   }else{
      return __int_as_float(c+0x4b000000)-8388608.f;
   }
}

__portable__ float fastfloats(byte c){
   if(targeting("CUDA")){
      return (float)(int)(char)c;
   }else{
      int z=(int)(uint)c;
      return __int_as_float(z^0x4b000080)-8388736.f;
   }
}

__portable__ int fastintus(float f){
   if(targeting("CUDA")){
      return __float2int_rn(f)-128;
   }else{
      return __float_as_int(f+8388608.f)-0x4b000080;
   }
}

__portable__ int fastints(float f){
   if(targeting("CUDA")){
      return __float2int_rn(f);
   }else{
      return __float_as_int(f+8388736.f)^0x4b000080;
   }
}

__portable__ int fastint16s(float f){
   if(targeting("CUDA")){
      return __float2int_rn(f);
   }else{
      return __float_as_int(f+8421376.f)^0x4b008000;
   }
}

void makeYuvBlock(auto py,auto pu,auto pv,auto img,
      int idelta,int bbase,int nb16,int w,int h){
   auto nb8=nb16*6;
   auto wb=(w+15)>>4,hb=(h+15)>>4;
   //produce UV in the Y block order
   forall([ofs,xb,yb] in makePartialGrid(
         bbase*256,(bbase+nb16)*256,
         256,wb,hb)){
      auto x=xb*16+(ofs&7)+((ofs&64)>>3);
      auto y=yb*16+((ofs>>3)&7)+((ofs&128)>>4);
      auto pc=idelta+(min(y,h-1)*w+min(x,w-1))*3;
      auto r=fastfloatu((int)img[pc+2]);
      auto g=fastfloatu((int)img[pc+1]);
      auto b=fastfloatu((int)img[pc]);
      auto Y=0.299f*r+0.587f*g+0.114f*b;
      py.push_back((byte)fastintus(Y));
      pu.push_back((byte)fastints(0.56433f*(b-Y)));
      pv.push_back((byte)fastints(0.71326f*(r-Y)));
   }
}

//round-to-nearest integer /4
__portable__ int div4(int c){
   c+=(c>>31)<<2;
   c+=2;
   return c>>2;
}

byte<> downSampleBlock(byte<> pu){
   auto puh=new byte<>;
   forall(i=0:pu.n/4-1){
      auto pos=((i>>6)*4+(((i>>2)&1)+((i>>4)&2)))*64+
         (i&(3*8+3))*2;
      int c=0;
      c+=(int)(char)pu[pos];
      c+=(int)(char)pu[pos+1];
      c+=(int)(char)pu[pos+8];
      c+=(int)(char)pu[pos+9];
      puh.push_back((byte)div4(c));
   }
   return puh;
}

//Compute DC components of one block from CPU
int3 makeLastBlock(auto img,int w,int h,
      int xlast,int ylast){
   char Ys[256];
   char us[256];
   char vs[256];
   for(int i=0;i<256;i++){
      auto x=(i&15),y=(i>>4);
      x=min(xlast+x,w-1);
      y=min(ylast+y,h-1);
      auto pc=&img[(y*w+x)*3];
      auto r=fastfloatu((int)pc[2]);
      auto g=fastfloatu((int)pc[1]);
      auto b=fastfloatu((int)pc[0]);
      auto Y=0.299f*r+0.587f*g+0.114f*b;
      Ys[i]=((char)fastintus(Y));
      us[i]=((char)fastints(0.56433f*(b-Y)));
      vs[i]=((char)fastints(0.71326f*(r-Y)));
   }
   int ytot=0,utot=0,vtot=0;
   for(int i=0;i<64;i++){
      auto x=(i&7),y=(i>>3);
      int puv=x*2+(y*2)*16;
      ytot+=(int)Ys[x+8+(y+8)*16];
      utot+=div4((int)us[puv]+(int)us[puv+1]+
            (int)us[puv+16]+(int)us[puv+17]);
      vtot+=div4((int)vs[puv]+(int)vs[puv+1]+
            (int)vs[puv+16]+(int)vs[puv+17]);
   }
   return make_int3(
      fastint16s((float)ytot*t_Y[0]),
      fastint16s((float)utot*t_Cb[0]),
      fastint16s((float)vtot*t_Cb[0]))
}

__device__ void DCT8(float* a,int pitch){
   float tmp0,tmp1,tmp2,tmp3,tmp4,tmp5,tmp6,tmp7;
   float tmp10,tmp11,tmp12,tmp13;
   float z1, z2, z3, z4, z5, z11, z13;
   tmp0 = a[pitch*0] + a[pitch*7];
   tmp7 = a[pitch*0] - a[pitch*7];
   tmp1 = a[pitch*1] + a[pitch*6];
   tmp6 = a[pitch*1] - a[pitch*6];
   tmp2 = a[pitch*2] + a[pitch*5];
   tmp5 = a[pitch*2] - a[pitch*5];
   tmp3 = a[pitch*3] + a[pitch*4];
   tmp4 = a[pitch*3] - a[pitch*4];

   tmp10 = tmp0 + tmp3;   /* phase 2 */
   tmp13 = tmp0 - tmp3;
   tmp11 = tmp1 + tmp2;
   tmp12 = tmp1 - tmp2;

   a[pitch*0] = tmp10 + tmp11; /* phase 3 */
   a[pitch*4] = tmp10 - tmp11;

   z1 = (tmp12 + tmp13) * ((float) 0.707106781);
   a[pitch*2] = tmp13 + z1;   /* phase 5 */
   a[pitch*6] = tmp13 - z1;

   tmp10 = tmp4 + tmp5;   /* phase 2 */
   tmp11 = tmp5 + tmp6;
   tmp12 = tmp6 + tmp7;

   z5 = (tmp10 - tmp12) * ((float) 0.382683433);
   z2 = ((float) 0.541196100) * tmp10 + z5;
   z4 = ((float) 1.306562965) * tmp12 + z5;
   z3 = tmp11 * ((float) 0.707106781);

   z11 = tmp7 + z3;      /* phase 5 */
   z13 = tmp7 - z3;

   a[pitch*5] = z13 + z2;   /* phase 6 */
   a[pitch*3] = z13 - z2;
   a[pitch*1] = z11 + z4;
   a[pitch*7] = z11 - z4;

}

struct CDctCoefficient{
   short a[64];
};
void dctQuantitize(CDctCoefficient<> ret,int rbase,
byte<> a,int dcpre,int dclast,float[64] tab){
   int nb;
   nb=a.n>>6;
   auto dc=new short<nb+1>;
   forall(i=0:nb-1){
      float d[64];
      auto ib=i*64;
      For(j=0:63){
         d[j]=fastfloats(a[ib+j]);
      }
      For(j=0:7){
         DCT8(d+j*8,1);
      }
      For(j=0:7){
         DCT8(d+j,8);
      }
      int ri=rbase+i;
      For(j=0:63){
         const{j2=(int)zigzag[j];}
         int q=fastint16s(d[j]*tab[j]);
         if(j==0){
            dc[i+1]=(short)q;
         }
         ret[ri].a[j2]=(short)q;
      }
   }
   dc[0]=dcpre;
   ret[rbase+nb-1].a[0]=dclast;
   forall(i=0:nb-1){
      ret[rbase+i].a[0]-=dc[i];
   }
}

__device__ void getCategoryBitcode(
      int& category,int& bitcode,int a){
   if(targeting("CUDA")){
      int fi=__float_as_int((float)a);
      category=((fi>>23)&0xff)-0x7e;
   }else{
      int ap=(int)a;
      if(ap<0){ap=-ap;}
      category=1;
      if(targeting("x86")){
         category+=_BitScanReverse(ap);
      }else{
         if(ap>=256){ap>>=8;category+=8;}
         if(ap>=16){ap>>=4;category+=4;}
         if(ap>=4){ap>>=2;category+=2;}
         if(ap>=2){/*ap>>=1;*/category+=1;}
      }
   }
   bitcode=(int)a;
   if(bitcode<0)bitcode+=(1<<category)-1;
}

const int HASH_DC=0;
const int HASH_AC=1;

//RLE and Huffman encoding
__device__ int encodeBlock(
      byte<> huffman,
      int& nbit,int& bits,
      auto dct,int bid,int isUV,
      int<> lgs,int<> codes){
   int total=0;
   isUV*=(16+256);
   auto bitsWriter=[](int cat,int sym){
      total+=cat;
      nbit+=cat;
      bits=(bits<<cat)+sym;
      For(i=0:1){
         if(nbit>=8){
            nbit-=8;
            huffman.push_back((byte)(bits>>nbit));
         }
      }
   };
   auto huffManWriter=[](int side,int sym){
      int hsym=side*16+isUV+sym;
      bitsWriter(lgs[hsym],codes[hsym]);
   };
   int Diff=(int)dct[bid].a[0];
   int category,bitcode;
   if (Diff == 0){
      huffManWriter(HASH_DC,0); //Diff might be 0
   }else{
      getCategoryBitcode(category,bitcode,Diff);
      huffManWriter(HASH_DC,category);
      bitsWriter(category,bitcode);
   }

   // Encode ACs
   int nz=0;
   for(int i=1;i<64;i++){
      int c=(int)dct[bid].a[i];
      if(c==0){
         nz++;
      }else{
         For(j=0:2){
            if(nz>=16){
               huffManWriter(HASH_AC,0xf0);
               nz-=16;
            }
         }
         getCategoryBitcode(category,bitcode,c);
         huffManWriter(HASH_AC,nz*16+category);
         bitsWriter(category,bitcode);
         nz=0;
      }
   }
   if(nz)huffManWriter(HASH_AC,0);
   return total;
}

class chuffmantab{
   int<> codes;
   int<> lgs;
   byte* syms;
   int nsym;
   void __init__(byte* nrcodes,byte* values,int n){
      this.nsym=n;
      this.syms=new byte[n+16];
      memcpy(this.syms,nrcodes+1,16);
      memcpy(this.syms+16,values,n);
      //make huffman table
      struct clengthid{
         int lg;
         int id;
      };
      auto nx= n==12?16:256;
      this.lgs=new int<nx>;
      this.codes=new int<nx>;
      auto lgsrt=new clengthid[n];
      auto p=0;
      for(int lg=1;lg<=16;lg++){
         auto nlg=(int)nrcodes[lg];
         for(int i=0;i<nlg;i++){
            auto id=(int)values[p];
            lgsrt[p].lg=lg;
            lgsrt[p].id=id;
            this.lgs[id]=lg;
            p++;
         }
      }
      auto clg=0,ccode=0;
      for(int i=0;i<n;i++){
         auto lgi=lgsrt[i].lg;
         if(!lgi)continue;
         while(clg<lgi){
            clg++;
            ccode+=ccode;
         }
         this.codes[lgsrt[i].id]=ccode;
         ccode++;
      }
      delete lgsrt;
   }
   __done__(){
      if(this.syms)delete this.syms;
   }
};

inline void writeBuf(byte*& pjpeg,void* buf,int n){
   memcpy(pjpeg,buf,n);
   pjpeg+=n;
}

byte<> compactHuffman(auto huffman0, auto outofs,
auto totbits,auto inofs,auto nbithuff){
   auto huffman=new byte< (nbithuff+7)>>3 >;
   forall(pout in outofs with
         total in totbits, pin in inofs){
      int nbshift=-pout&7;
      pout+=nbshift;
      pout>>=3;
      int nmybit=total-nbshift;
      int nbfill=((nmybit+7)>>3)-1;
      int p=pin;
      for(int j=0;j<nbfill;j++){
         huffman[pout++]=(byte)((
            (int)huffman0[p]<<nbshift)+
            ((int)huffman0[p+1]>>(8-nbshift)));
         p++;
      }
      if(nbfill>=0){
         //tail byte
         int nbit=nmybit-(nbfill<<3);
         int bits=(int)huffman0[p]<<nbshift;
         if(nbit>(8-nbshift))
            bits+=(int)huffman0[p+1]>>(8-nbshift);
         bits>>=(8-nbit);
         int ptotbits=__index+1;
         while(nbit<8&&ptotbits<totbits.n){
            int nbnext=min(totbits[ptotbits],8);
            nbit+=nbnext;
            bits=(bits<<nbnext)+((int)huffman0[
               inofs[ptotbits]]>>(8-nbnext));
            ptotbits++;
         }
         //last byte case
         if(nbit<8){
            bits<<=(8-nbit);
         }else{
            bits>>=nbit-8;
         }
         huffman[pout++]=(byte)bits;
      }
   }
   return huffman;
}
```

```
int rleAndHuffman(byte<> huffman,auto dct,auto nb,
auto lgsAll,auto codesAll){
   auto totbits=new int<>;
   auto inofs=new int<>;
   auto p_nbit=new CPersistentVariable(int)(0);
   auto p_bits=new CPersistentVariable(int)(0);
   auto p_total=new CPersistentVariable(int)(0);
   auto nbithuff=0;
   forall"novector,nomeasure"(
         [what,bid] in makeGrid(6,nb)){
      auto b;
      if(what<4){
         b=bid*4+what;
      }else{
         b=bid+nb*what;
      }
      int nbit=p_nbit.value, bits=p_bits.value;
      int total=encodeBlock(huffman,
         nbit,bits,
         dct,b,what>>2,
         lgsAll,codesAll);
      p_nbit.value=nbit;
      p_bits.value=bits;
      p_total.value+=total;
      if(targeting("CUDA")){
         //   On GPU, we have to compact per-block
         //huffman after this pass.
         if(nbit>0){
            bits<<=(8-nbit);
            huffman.push_back((byte)bits);
         }
         totbits.push_back(total);
         inofs.push_back((total+7)>>3);
      }
   }
   if(p_total.value!=0){
      int endnbit=p_nbit.value;
      int endbits=p_bits.value;
      nbithuff=p_total.value;
      if(endnbit>0){
         endbits<<=(8-endnbit);
         huffman.push_back((byte)endbits);
      }
   }else{
      //compact per-block huffman bits for GPU
      auto outofs=new int<totbits.n>;
      nbithuff=scan(rop_add,outofs,totbits);
      scan(rop_add,inofs,inofs);
      auto huffman0=huffman;
      huffman=compactHuffman(huffman0,outofs,
         totbits,inofs,nbithuff);
   }
   return nbithuff;
}

byte<> encodeJpeg(byte* pimg,int w,int h,int quality){
   auto jpeg=new byte<>;
   jpeg.storageSide=STORE_CPU;
   SOF0info.width=wordSwap(w);
   SOF0info.height=wordSwap(h);
   initDQT((BYTE)quality);
   auto hddcY=new chuffmantab(
      std_dc_luminance_nrcodes,
      std_dc_luminance_values,12);
   auto hdacY=new chuffmantab(
      std_ac_luminance_nrcodes,
      std_ac_luminance_values,162);
   auto hddcUV=new chuffmantab(
      std_dc_chrominance_nrcodes,
      std_dc_chrominance_values,12);
   auto hdacUV=new chuffmantab(
      std_ac_chrominance_nrcodes,
      std_ac_chrominance_values,162);
   auto lgsAll=new int<>;
   auto codesAll=new int<>;
   lgsAll.add(hddcY.lgs);
   lgsAll.add(hdacY.lgs);
   lgsAll.add(hddcUV.lgs);
   lgsAll.add(hdacUV.lgs);
   codesAll.add(hddcY.codes);
   codesAll.add(hdacY.codes);
   codesAll.add(hddcUV.codes);
   codesAll.add(hdacUV.codes);
   //file header
   DHTinfo.length=4+16*4+(12+162)*2+2;
   jpeg.resize(sizeof(APP0info)+sizeof(DQTinfo)+
      sizeof(SOF0info)+2+(int)DHTinfo.length+
      sizeof(SOSinfo));
   DHTinfo.length=wordSwap((int)DHTinfo.length);
   auto pjpeg=&jpeg[0];
   #define writeBig(buf) \
      memcpy(pjpeg,&buf,sizeof(buf));\
      pjpeg+=sizeof(buf)
   writeBig(APP0info);
   writeBig(DQTinfo);
   writeBig(SOF0info);
   writeBig(DHTinfo);
   writeBig((byte)0x00);
   writeBuf(pjpeg,hddcY.syms,hddcY.nsym+16);
   writeBig((byte)0x10);
   writeBuf(pjpeg,hdacY.syms,hdacY.nsym+16);
   writeBig((byte)0x01);
   writeBuf(pjpeg,hddcUV.syms,hddcUV.nsym+16);
   writeBig((byte)0x11);
   writeBuf(pjpeg,hdacUV.syms,hdacUV.nsym+16);
   writeBig(SOSinfo);
   assert(pjpeg-&jpeg[0]==jpeg.n);
   #undef writeBig
   //encoding starts
   auto winb=(w+15)>>4;
   auto hinb=(h+15)>>4;
   auto nbtot=winb*hinb;
   int nbittotal=0,nbittar=0;
   lgsAll.broadcast();
   codesAll.broadcast();
   distribute(b0:b1 in 0:nbtot-1 step [1<<8, 1<<14]){
      auto y0=b0/winb,x0=b0-y0*winb;
      auto y1=b1/winb,x1=b1-y1*winb;
      int base;
      //Cross-processor boundary handling:
      //    Recompute first and last block's DC
      //  components from CPU to hide precision
      //  discrepancy.
      int3 dcpre=make_int3(0,0,0);
      if(b0>0){
         auto y0pre=(b0-1)/winb;
         auto x0pre=(b0-1)-y0pre*winb;
         auto xpre=x0pre*16,ypre=y0pre*16;
         dcpre=makeLastBlock(pimg,w,h,xpre,ypre);
      }
      int3 dclast=makeLastBlock(pimg,w,h,x1*16,y1*16);
      auto ptrbase=((y0*16)*w+x0*16)*3;
      auto img=new byte<>;
      base=img.mount(pimg+ptrbase,
         (min(y1*16+15,h-1)*w+min(x1*16+15,w-1)+1)*3-
         ptrbase);
      //RGB to YCbCr
      auto py=new byte<>;
      auto pu=new byte<>;
      auto pv=new byte<>;
      makeYuvBlock(py,pu,pv, img,base-ptrbase,
         b0,b1+1-b0, w,h);
      //CbCr downsampling
      auto puh=downSampleBlock(pu); delete pu;
      auto pvh=downSampleBlock(pv); delete pv;
      int nb=b1+1-b0;
      //DCT and quantitize
      auto dct=new CDctCoefficient<nb*6>;
      dctQuantitize(dct,0,py,dcpre.x,dclast.x,t_Y);
      delete py;
      dctQuantitize(dct,nb*4,puh,dcpre.y,dclast.y,t_Cb);
      delete puh;
      dctQuantitize(dct,nb*5,pvh,dcpre.z,dclast.z,t_Cb);
      delete pvh;
      //RLE and Huffman encoding
      auto huffman=new byte<>;
      auto nbithuff=rleAndHuffman(huffman,dct,nb,
         lgsAll,codesAll);
      if(b1==nbtot-1&&(nbithuff&7)!=0){
         //one-bits fill for last block
         huffman[huffman.n-1]|=
            (byte)(1<<(-nbithuff&7)-1);
      }
      img.unmount();
      serialize{
         nbittotal+=nbithuff;
      if(spap.isLastTask){
         auto szreserve=(int)(
            (float)((nbittotal+7)>>3)*1.05f);
         jpeg.reserve(jpeg.n+szreserve);
      }
   }
   serialize{
      auto nbshift=-nbittar&7;
      auto nblast=8-(-nbithuff&7);
      if(nbshift){
         auto blast=(jpeg[jpeg.n-1]|=
            huffman[0]>>(8-nbshift));
         if(blast==(byte)0xff){
            jpeg.push_back((byte)0x00);
         }
         forall"novector,nomcore"(
               i=0:huffman.n-2){
            auto b=(huffman[i]<<nbshift)+
               (huffman[i+1]>>(8-nbshift));
            jpeg.push_back(b);
            if(b==(byte)0xff){
               jpeg.push_back((byte)0x00);
            }
         }
         //last byte
         if(nbshift<nblast){
            jpeg.push_back(
               huffman[huffman.n-1]<<nbshift);
         }
      }else{
         forall"nosse,nomcore"(b in huffman){
            jpeg.push_back(b);
            if(b==(byte)0xff){
               jpeg.push_back((byte)0x00);
            }
         }
      }
      nbittar+=nbithuff;
   }
}
jpeg.push_back((byte)0xff);
jpeg.push_back((byte)0xd9);
return jpeg;
}

///////////////////////////////////////////////////
long long tbegin(){
   long long t0;
   spapFlush();
   QueryPerformanceCounter((LARGE_INTEGER*)&t0);
   return t0;
}

double tend(long long t0){
   long long t1,freq;
   spapFlush();
   QueryPerformanceCounter((LARGE_INTEGER*)&t1);
   QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
   return (double)(t1-t0)/(double)freq;
}

int main(int argc,char** argv){
   int w=0,h=0;
   auto bmp=new byte<>;
   auto qual=50;
   if(argc<=1){
      return 0;
   }else{
      auto f=fopen(argv[1],"rb");
      if(!f){
         printf("unable to open bmp %s\n",argv[1]);
         return 0;
      }
      fseek(f,0x12,SEEK_SET);
      fread(&w,sizeof(w),1,f);
      fread(&h,sizeof(h),1,f);
      fseek(f,0x36,SEEK_SET);
      bmp.resize(w*h*3);
      auto pbmp=&bmp[0];
      for(int i=0;i<h;i++){
         auto pline=pbmp+(h-1-i)*w*3;
         fread(pline,3*w,1,f);
         auto alg=(-3*w)&3;
         if(alg){
            fseek(f,alg,SEEK_CUR);
         }
      }
      fclose(f);
      if(argc>=3){
         sscanf(argv[2],"%d",&qual);
      }
   }
   //do the encoding
   auto f=fopen("!out.jpg","wb");
   auto t0=tbegin();
   auto jpeg=encodeJpeg(&bmp[0],w,h,qual);
   auto pj=jpeg.apiSafeMap(map_CPU|map_read);
   auto th1=tbegin();
   fwrite(pj,1,jpeg.n,f);
   auto tio=tend(th1);
   auto t=tend(t0);
   fclose(f);
   printf("I/O time: %.2lfms\n",tio*1000.);
   printf("Encoding time: %.2lfms\n",t*1000.);
   return 0;
}
```