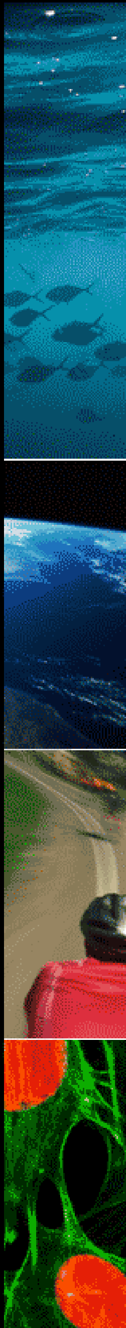
The Supercomputer Company

Cray X1 Compiler Challenges (And How We Solved Them)

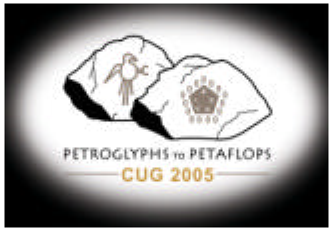


Terry Greyzck, Cray Inc.

CUG 2005

The Cray X1 Architecture

- First all-new vector architecture from Cray since the Cray-2 in 1985
- Incorporates experience from vector and MPP lines
- An all-new architecture comes with all-new challenges
- Compiler is relied upon to obtain maximum performance from the new machine



New Hardware Challenges

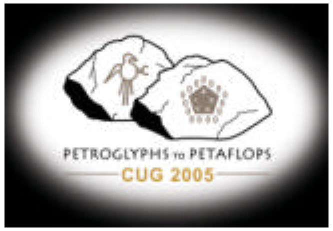
- New instruction set architecture (ISA)
- Hardware support for 32- and 64-bit operations
- The multistreaming processor (MSP)
- Decoupled scalar and vector operations
- Predicated vector execution
- These and other hardware changes provide the bulk of compiler challenges necessary for a successful Cray X1 compiler

New Software Challenges

- 8- and 16-bit integer support
- Structure packing and alignment compatible with other major vendors
- IEEE floating point support
- OpenMP for C and C++
- Unified Parallel C (UPC)
- These are in addition to features already supported on earlier machines

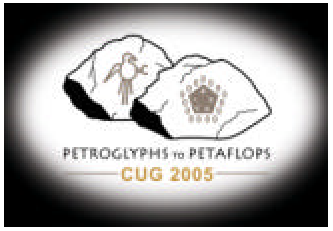
Differences from the Cray SV1

- New instruction set architecture
- Multistreaming processor
- Decoupled vector and scalar
- Predicated vector execution
- Multi-level cache
- Globally addressable memory
- IEEE floating point



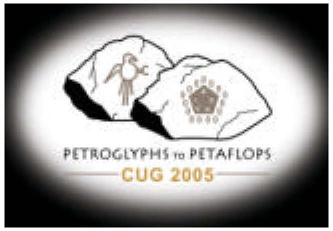
Levels of Parallelism

- The compiler automatically extracts:
  - Multiple registers
  - Multiple functional unit groups
  - Vectorization
  - Multistreaming
- The user has to help with:
  - Higher-level programming models, such as CoArray Fortran, UPC, and OpenMP



Compiler Requirements

- Correctness and stability
- Automatic multistreaming
- Improved automatic vectorization
- Support for parallel programming models
- Support for industry-standard data models

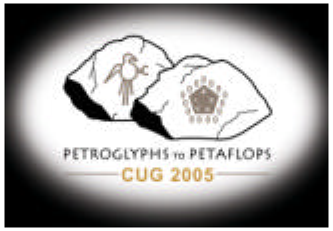


Compiler Front Ends

- Lower source code to an intermediate-level representation
- Fortran front end developed originally by Cray Research
- C and C++ use a commercial front end, modified to support Cray extensions
- Communicate with other components using a functional interface

Interprocedural Optimizer

- New component for the Cray X1; replaces older language-specific inliners
- Primarily provides support for inlining, including cross-file inlining
- Supports procedure cloning
- Provides other optimizations such as tail recursion elimination and Fortran alternate entry point simplification



PDGCS Optimizer

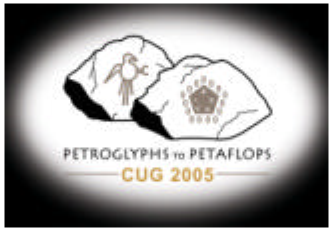
- Initially developed for earlier vector architectures
- Continually updated to meet performance and functional requirements
- Provides core optimization and parallelization capabilities for the compiler
- Vectorization, multistreaming happen here
- Also performs ‘scalar’ optimizations

Code Generator

- Written from scratch for the Cray X1
- Performs final lowering of intermediate level text into assembly code
- Scheduler reorders intermediate text for overall performance
- Uses Elf and Dwarf representations for output

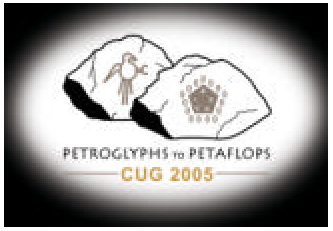
Compiler Challenge: Support for 32-bit Operations

- Previous architectures were very 64-bit oriented
- Cray X1 has hardware support for 32-bit operations
- Adding compiler support for this was tedious
- Some problems were encountered, mostly dealing with mixed-size expressions



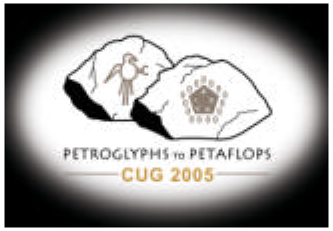
Support for 16-bit Operations

- No Cray compiler ever supported 16-bit integers before the Cray X1
- No direct hardware support, and limited set of helper instructions
- Most of the 8- and 16-bit work is done in software, and uses 64-bit loads and stores
- Vectorization of 8- and 16-bit loads is supported
- Fortran maps one and two byte integers to four byte integers by default, for performance



IEEE Floating-Point Support

- The Cray T90 and Cray T3E had IEEE floating point (on the Cray T90 it was an option)
- Most of the compiler work done for IEEE on those platforms was reused
- Created the –*Ofp**n*** option for better control over floating point optimization
- Higher levels of the –*Ofp**n*** option allow the compiler greater freedom to optimize floating point expressions

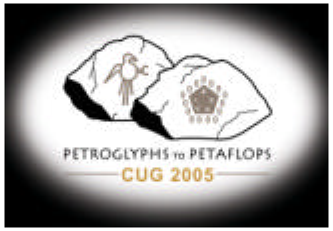


32 Vector Registers

- Earlier architectures only had eight
- Each vector register has 64 elements
- This provides a 32x64 = 2,048 element cache for vector operations
- The additional vector registers are used as cache when possible
- Traditional optimizations were also modified to work with vectors, such as invariant hoisting and strength reduction

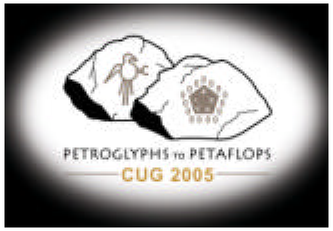
Predicated Vector Execution

- Used for conditionally executed vector code
- The conditional test creates a vector mask result
- That mask result controls what expressions are evaluated, on an element by element basis
- Helpful in removing scalar test and branch code, and in unrolling vector loops
- A zero vector length can also be used for predicated execution in some cases



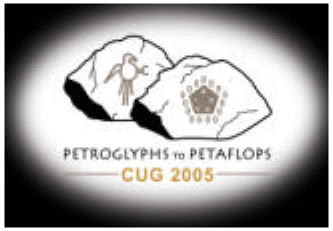
Decoupled Vector and Scalar Operations

- Synchronization between vector and scalar memory operations is a software responsibility on the Cray X1
- The challenge is to create the least expensive synchronization possible while maintaining correctness
- Vector region expansion reduces synchronization costs further



Longer Memory Latencies

- Memory keeps getting further away...
- Hardware provides some latency covering
- Scheduling is the primary software method for covering latency
- Increasing the basic block size (within reason) by various transformations allows the scheduler more freedom to push loads backwards
- Software vector pipelining is also very useful in some cases; as much as a 30% improvement



Automatic MSP Parallelism

- Single largest compiler development effort for the Cray X1
- Starts where Cray SV1 multistreaming left off
- Automatic shared memory parallelism
- The Cray X1 has significant hardware support for multistreaming, including:
  - Coherent caches
  - Synchronization primitives

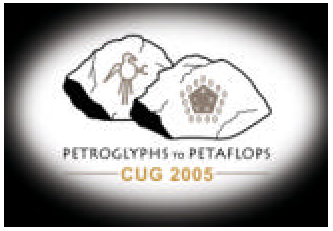
PETROGLYPHS to PETAFLOPS
CUG 2005

Multistreaming Constructs

- Multistreaming concentrates on loop nests, vectorizing one level and partitioning another among the processors
- Repeatability of results was a critical design factor
- Some things not generally thought of as parallel multistream well
- For a single loop, can make four two-pipe vector processors effectively behave as one eight-pipe vector processor

Minimizing Synchronization Overhead

- Even with hardware support, synchronization is costly
- Streamed region expansion extends the parallel section of code beyond loop boundaries
- Entire procedures can be multistreamed in this fashion
- For any synchronization that is still required, the least costly method is used



Minimizing Negative Optimization Effects

- Synchronization barriers can degrade performance throughout a function, not just in the parallel region
- Hard barriers are deferred until very late, allowing optimization freedom to move code past the synchronization points
- A final phase corrects any data scoping issues created by this

Balancing SIMD and MIMD Parallelism

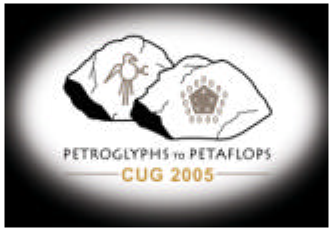
- Balancing vector and multistream parallelism is very difficult to do well
- The compiler attempts to use *both* vector and multistream parallelism
- Vector parallelism is more important than multistream parallelism for raw performance
- The correct use of loop transformations can maximize the performance extracted from each

Inner Vector, Outer Multistreamed

- The most common configuration for a loop nest
- One of the inner loops is vectorized
- One of the outer loops is multistreamed
- Loop transformations can help rewrite loops to take the best advantage
- Stand-alone loops are both vectorized and multistreamed

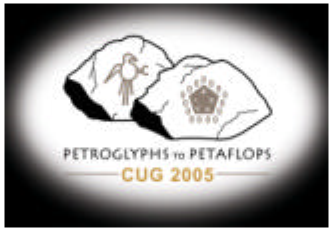
Inner Multistreamed, Outer Vector

- One of the outer loops is vectorized, and runs redundantly on all processors of the MSP
- An inner loop is partitioned across the processors
- This method allows for good use of the vector registers as very fast cache
- For common *dgemm* matrix multiplications, this technique allows us to achieve 99% of theoretical peak performance



Giant Applications!

- Application functions and subroutines keep getting bigger and more demanding
- As compiler developers, we continue to be surprised as limits are exceeded
- 10,000 vector loops in one subroutine
- 1,000,000 unique expressions in one function
- Exponentially increasing application size
- Compiler regioning addresses some of the issues



Conclusion

- The X1 architecture provides significant functional and optimization challenges for a compiler
- Trends in application complexity also are a challenge for the compiler
- New optimization technology was created to address these issues
- Development continues to further solidify the abilities of the compiler