



Concise parallelism



Natural C/C++ Parallelism

A single operator to control **multiple parallel programming paradigms**

```
void salute()  
{  
    parallel()  
    {  
        int idx = pix();  
        serial()  
        {  
            parallel(3)  
            {  
                printf("Hello, world, from task %d-%d\n", idx, pix());  
            }  
        }  
    }  
}
```

A single operator to control parallel synchronization

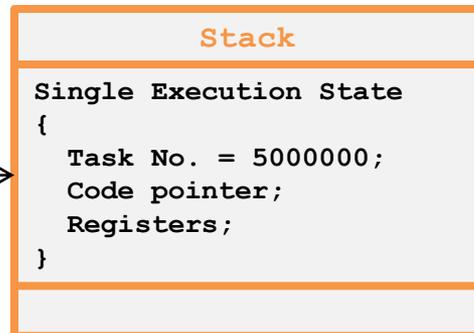
Natural C/C++ semantics and **variable visibility rules** and scopes

Clear means of parallel identification and interaction

Elegant Multitasking

```
std::vector<Data> data;  
  
parallel(5000000)  
{  
    int i = pix();  
  
    serial(&data[i])  
    {  
        data[i].process();  
    }  
}
```

Synchronized access to any data element without introducing synchronization objects



Each thread from a pool decrements the task counter and “creates” a job to execute from a single execution state:

- No CPU oversubscription
- Dynamic work balancing
- Minimal memory footprint
- No task queue management overhead

Language-Friendly Multithreading

A real independent thread in a class constructor!

```
class X
{
    void* volatile id;

    X()
    {
        parallel(2)
        {
            void* pid = pid();

            if(pid())
            {
                id = pid;

                while(id)
                {
                    wait();
                    getMoreData();
                }
            }
            break;
        }
    }
};
```

A single operator to control multi-threading and multitasking

Getting a global ID promotes a task to an independent thread

Thread-0 returns, thread-1 waits until woken up by another thread/task

Reaching the break demotes a thread to a task

```
void X::read()
{
    wake(id);

    processData();
}
```

Easy Software Analysis

Use the same **compiler, debugger and profiler** tools as for sequential software

```
std::vector<Data> data;

void f(int n)
{
    parallel(data.size)
    { /// Timing: 5 sec; Parallelism = 95%; Time per CPU: CPU0 = 30%, CPU1 = 30%...
        for(int i = 0; i < n; i++)
        { /// Avrg iterations = 100
            int j = pix();

            parallel()
            { /// Timing: 4.5 sec; Parallelism = 80%; Time per CPU: CPU0 = 30%, CPU1 = 30%...
                data[j].process();

                serial()
                { /// Timing: 4.5 sec; Contention = 30%;
                    data[j].reduce();
                }
            }
        }
    }
}
```

C= source code is a perfect performance model by itself: a C= profiler can annotate each parallel, sequential and cyclic region with timings, contention, iterations, balance, etc. exactly in alignment with a corresponding operator

Software Implications

A powerful parallel programming language...

C =

...and a unified parallel runtime

OpenMP

TBB

Cilk

CRT

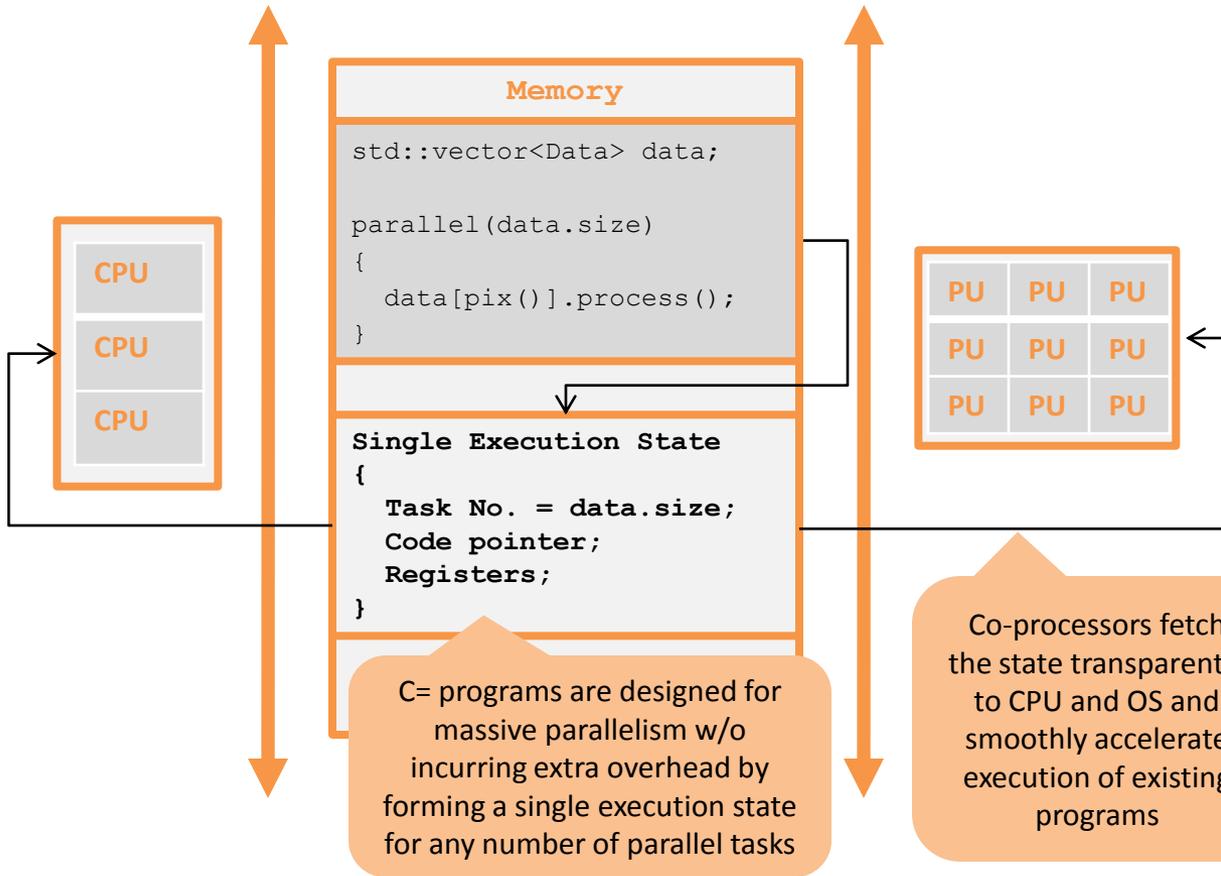
PPL

AMP @CPU

OpenCL @CPU

Re-writing parallel runtimes in C= will eliminate CPU oversubscription and guarantee efficient resource management, especially in complex, multi-module applications using several parallel runtimes simultaneously

Hardware Implications

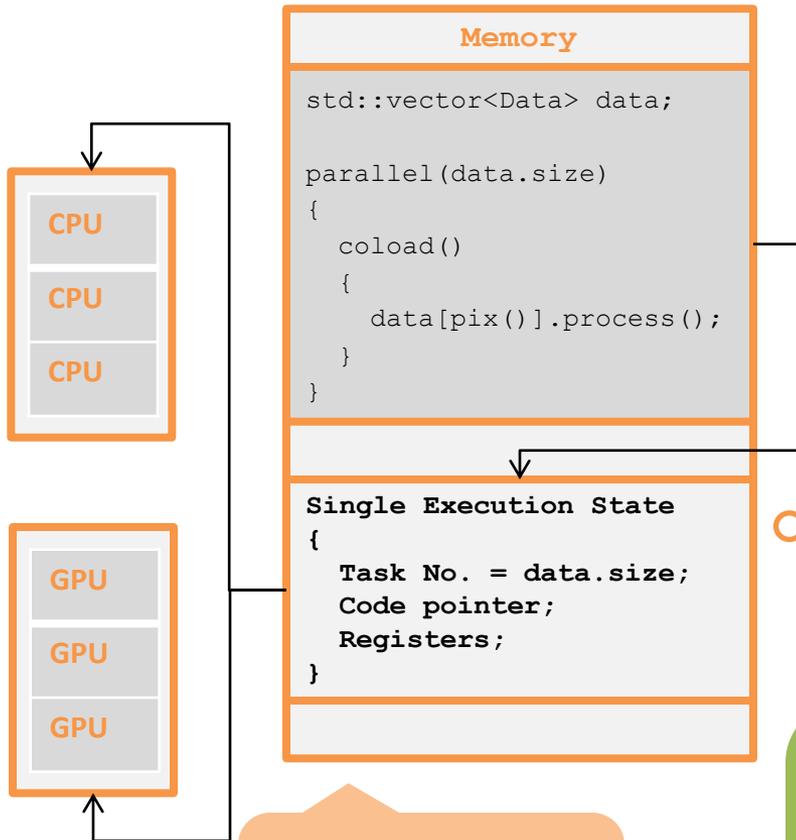


Slide a tablet into an accelerator box and get faster software, vivid graphics, detailed scenes, real-time video encoding – right away!



Truly mobile, data-consistent, cheap and powerful architecture!

One Program Fits All



C= programs are executed concurrently by CPUs and GPUs



Remote agents may concurrently "steal" the work from C= execution states and utilize their CPUs and GPUs

Unified Semantic Concept of Parallelism enables distributed heterogeneous programming with a single parallel operator