Microarchitectural Analysis and Optimization Techniques

Gunther Huebler
Collaborators: Vincent Larson, John Dennis

All the Work Presented Has Been Implemented in CLUBB (Cloud Layers Unified By Binormals)

CLUBB is a model that solves a set of partial differential equations in height and time.

Usable as a standalone model or as a subgrid parameterization in large scale models.

Implemented by default in CAM (Community Atmosphere Model), and various other models.

CLUBB costs roughly 30% of CAM. Optimizing it can go a long way.

Outline

- Intel's VTune Amplifier is a powerful tool
- There are multiple ways to diagnose bottlenecks
- Code changes discussed here have significantly reduced the cost of CLUBB
- Intel's MKL_VML functions are quite versatile
- Lapack libraries are less efficient than compiling from source

VTune Amplifier is a Powerful Way to Analyze Code Performance

VTune Amplifier is a performance analysis tool developed by Intel.

It can utilize Performance Monitoring Units (PMUs) to provide hardware event-based sampling.

Code profiles include detailed hardware specific metrics:

- Scalar/Vector/Division instruction counts
- Counts of stalls due to L(1/2/3) cache misses
- Branch Clears

Exploration modes include hotspots and tree breakdowns.

Using VTune to Analyze Polynomial Calculation

Consider an 8th degree polynomial: $a_9x^8+a_8x^7+a_7x^6+a_6x^5+a_5x^4+a_4x^3+a_3x^2+a_2x^1+a_1$

Compare: Horner's Method: $(((((((a_9x+a_8)x+a_7)x+a_6)x+a_5)x+a_4)x+a_3)x+a_2)x+a_1$

Custom Implementation: $((((a_9x+a_8)x^2+(a_7x+a_6))x^2+(a_5x+a_4))x^2+(a_3x+a_2))x+a_1$

Horner's method: Minimizes calculations, but has a large dependency chain

```
y = a(1) + x * ( a(2) + x * ( a(3) + x &
    * ( a(4) + x *( a(5) + x*( a(6) + x &
    * ( a(7) + x*( a(8) + x* a(9) ) ) ) ) ) ) )
```

Custom Implementation: Slightly more calculations required, but breaks up the dependency chain

```
x_sqd = x**2

y = ( ( ( ( a(9) * x + a(8) ) * x_sqd &
            + ( a(7) * x + a(6) ) ) * x_sqd &
          + ( a(5) * x + a(4) ) ) * x_sqd &
        + ( a(3) * x + a(2) ) ) * x + a(1)
```

VTune's Assembly Viewer, Instruction Count, Clocktick Metric, and CPI Rate

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ horner	1,318,000,000	3,234,000,000	0.408

Address ▲	So...	Assembly	Clockticks
0x402b8c		**Block 2:**	
0x402b8c	20	vmovapd %ymm11, %ymm3	400,000
0x402b91	20	vfmadd231pd %ymm2, %ymm12, %ymm3	147,600,000
0x402b96	20	vfmadd213pd %ymm10, %ymm2, %ymm3	400,000
0x402b9b	20	vfmadd213pd %ymm9, %ymm2, %ymm3	358,800,000
0x402ba0	20	vfmadd213pd %ymm8, %ymm2, %ymm3	800,000
0x402ba5	19	vfmadd213pd %ymm7, %ymm2, %ymm3	212,400,000
0x402baa	19	vfmadd213pd %ymm6, %ymm2, %ymm3	3,200,000
0x402baf	19	vfmadd213pd %ymm5, %ymm2, %ymm3	392,800,000
0x402bb4	19	vfmadd213pd %ymm4, %ymm3, %ymm2	201,600,000

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ custom	1,618,800,000	5,157,600,000	0.314

Address ▲	So...	Assembly	Clockticks
0x402f3c		**Block 1:**	
0x402f3c	87	vmovapd %ymm12, %ymm4	400,000
0x402f41	87	vmovapd %ymm10, %ymm15	160,400,000
0x402f46	85	vmulpd %ymm2, %ymm2, %ymm6	400,000
0x402f4a	87	vfmadd231pd %ymm2, %ymm13, %ymm4	344,000,000
0x402f4f	87	vfmadd231pd %ymm2, %ymm11, %ymm15	0
0x402f54	87	vfmadd213pd %ymm15, %ymm6, %ymm4	157,200,000
0x402f59	88	vmovapd %ymm8, %ymm15	2,800,000
0x402f5e	88	vfmadd231pd %ymm2, %ymm9, %ymm15	340,400,000
0x402f63	88	vfmadd213pd %ymm15, %ymm6, %ymm4	0
0x402f68	88	vmovapd %ymm1, %ymm15	196,000,000
0x402f6c	88	vfmadd231pd %ymm2, %ymm7, %ymm15	400,000
0x402f71	88	vfmadd213pd %ymm15, %ymm6, %ymm4	369,600,000
0x402f76	87	vfmadd213pd %ymm5, %ymm2, %ymm4	47,200,000

Clockticks are a simple way to compare performance.
The custom implementation is about **20% slower** than Horner's

Horner's method is able to use fewer operations by efficient use of fused multiply-add (FMA) instructions, but the long dependency chain hurts the clocks per instruction (CPI) rate.

How would these compare if compiled with -no-fma?

VTune Analysis Compiling with -no-fma

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ horner	3,985,200,000	6,505,600,000	0.613

Address ▲	So...	Assembly	Clockticks
0x402b8c		**Block 2:**	
0x402b8c	20	vmulpd %ymm10, %ymm0, %ymm12	400,000
0x402b91	20	vaddpd %ymm9, %ymm12, %ymm14	400,000
0x402b96	20	vmulpd %ymm14, %ymm0, %ymm15	38,400,000
0x402b9b	20	vaddpd %ymm8, %ymm15, %ymm12	474,000,000
0x402ba0	20	vmulpd %ymm12, %ymm0, %ymm14	400,000
0x402ba5	20	vaddpd %ymm7, %ymm14, %ymm12	800,000
0x402ba9	20	vmulpd %ymm12, %ymm0, %ymm15	42,000,000
0x402bae	20	vaddpd %ymm6, %ymm15, %ymm12	455,600,000
0x402bb2	20	vmulpd %ymm12, %ymm0, %ymm14	0
0x402bb7	19	vaddpd %ymm5, %ymm14, %ymm12	4,800,000
0x402bbb	19	vmulpd %ymm12, %ymm0, %ymm15	46,800,000
0x402bc0	19	vaddpd %ymm4, %ymm15, %ymm12	693,200,000
0x402bc4	19	vmulpd %ymm12, %ymm0, %ymm14	61,600,000
0x402bc9	19	vaddpd %ymm3, %ymm14, %ymm12	245,600,000
0x402bcd	19	vmulpd %ymm12, %ymm0, %ymm0	518,000,000
0x402bd2	19	vaddpd %ymm2, %ymm0, %ymm0	1,403,200,000

Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate
▶ custom	3,040,800,000	6,236,000,000	0.488

Address ▲	So...	Assembly	Clockticks
0x402f5d		**Block 1:**	
0x402f5d	87	vmulpd %ymm12, %ymm2, %ymm0	1,200,000
0x402f62	85	vmulpd %ymm2, %ymm2, %ymm3	94,000,000
0x402f66	87	vaddpd %ymm11, %ymm0, %ymm1	0
0x402f6b	87	vmulpd %ymm10, %ymm2, %ymm0	439,200,000
0x402f70	87	vmulpd %ymm3, %ymm1, %ymm1	400,000
0x402f74	87	vaddpd %ymm9, %ymm0, %ymm0	94,000,000
0x402f79	87	vaddpd %ymm0, %ymm1, %ymm1	0
0x402f7d	87	vmulpd %ymm3, %ymm1, %ymm0	410,800,000
0x402f81	88	vmulpd %ymm8, %ymm2, %ymm1	2,800,000
0x402f86	88	vaddpd %ymm7, %ymm1, %ymm1	75,600,000
0x402f8a	88	vaddpd %ymm1, %ymm0, %ymm0	0
0x402f8e	88	vmulpd %ymm3, %ymm0, %ymm0	411,200,000
0x402f92	88	vmulpd %ymm6, %ymm2, %ymm3	93,600,000
0x402f96	88	vaddpd %ymm5, %ymm3, %ymm1	143,200,000
0x402f9a	88	vaddpd %ymm1, %ymm0, %ymm0	0
0x402f9e	88	vmulpd %ymm2, %ymm0, %ymm2	534,800,000
0x402fa2	87	vaddpd %ymm4, %ymm2, %ymm0	740,000,000

Without FMA instructions, Horner’s method uses roughly the same number of operations. But now, it’s affected even more negatively by its dependency chain.

Compiled with -no-fma, the custom implementation is about **25% faster** than Horner’s.

The Custom Polynomial Reduces the Cost of CLUBB by 3%

CLUBB uses an 8th order polynomial to estimate saturation vapor pressure

- ''Polynomial Fits to Saturation Vapor Pressure'' Falatau, Walko, and Cotton. (1992) Journal of Applied Meteorology, Vol. 31, pp. 1507--1513

When compiled in CESM, the -no-fma option is used.

The custom method does not produce bit-for-bit identical results, but is mathematically equivalent.

Within CLUBB, the custom implementation was faster, regardless of compiler options.

VTune Can Diagnose the Expense of Library Functions

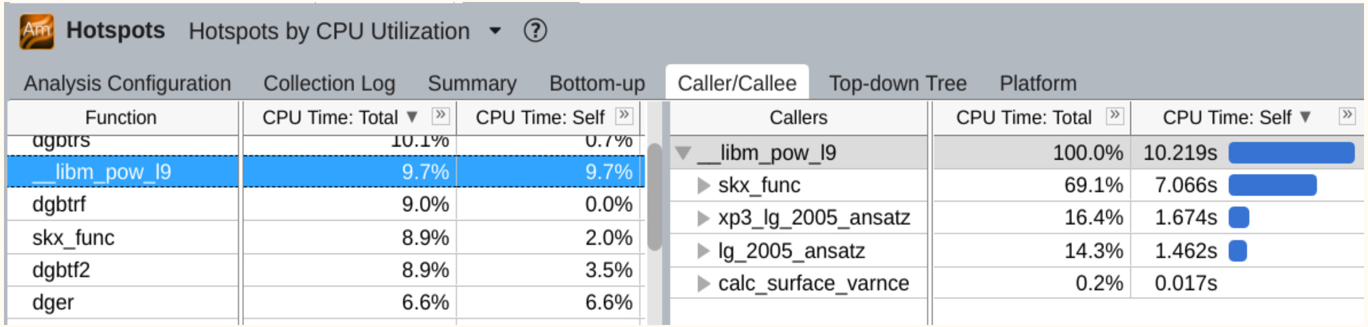
libm_pow_l9 is a library function used to calculate arbitrary floating point powers

- For example: 2^x, where x is some floating point value

We cannot optimize a library function, the only hope is to analyze the section of code which requires the use of such a function.

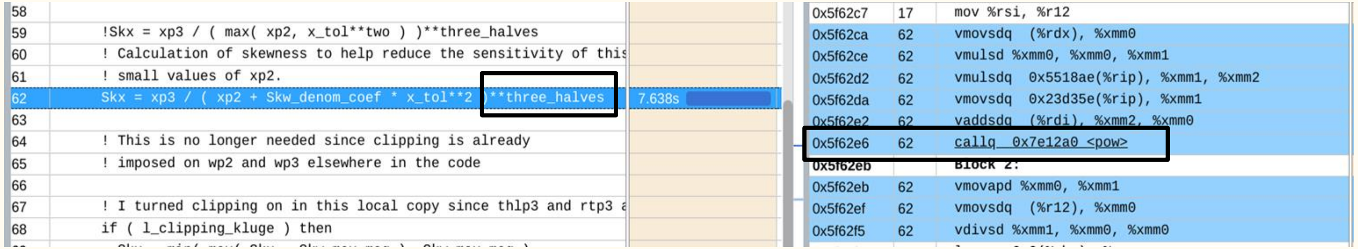
VTune's Caller/Callee breakdown within its hotspot analysis is a perfect tool to accomplish this.

Cost Analysis of libm_pow_l9



The caller/callee breakdown shows that the cost of libm_pow_l9 is coming from its use within the following functions:

- skx_func
- xp3_lg_2005_ansatz
- lg_2005_ansatz



Using the source/assembly viewer on one of these functions, we can find the exact bit of code where this function is used.

Now that we know the exact spot in code where this expense comes from, we can find a way to optimize.

Optimization of libm_pow_l9

The expense section of code has a constant power. More importantly the power is a multiple of 1/2.

```
Skx = xp3 / ( xp2 + Skw_denom_coef * x_tol**2 )**three_halves
```

Arbitrary powers can be expensive, but sqrt() functions are well optimized.

Using the equivalence **x^(3/2) = x * x^(1/2)**, we can refactor the code to become:

```
Skx = xp3 / ( ( xp2 + Skw_denom_coef * x_tol**2 ) * sqrt( xp2 + Skw_denom_coef * x_tol**2 ) )
```

sqrt() isn't cheap, but it is cheap relative to libm_pow_l9. This change produces bit-different results, but reduced overall runtime by ~10%.

Intel Has Special Vectorized Math Functions

Intel has a library that contains regular and special math functions, MKL_VML functions.

Many cover relatively simple functions:

- multiplication
- division
- powers and exponentials
- logarithms

There are also "special" math functions, which are particularly useful to CLUBB

- vdcdfnorm() computes the cumulative normal distribution function
- This replaces the need for the slow unvectorizable erf() function

$$\mathrm{cdfnorm}(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = 1 - \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Other functions also help to help index and copy values

- vdpack and vdunpack

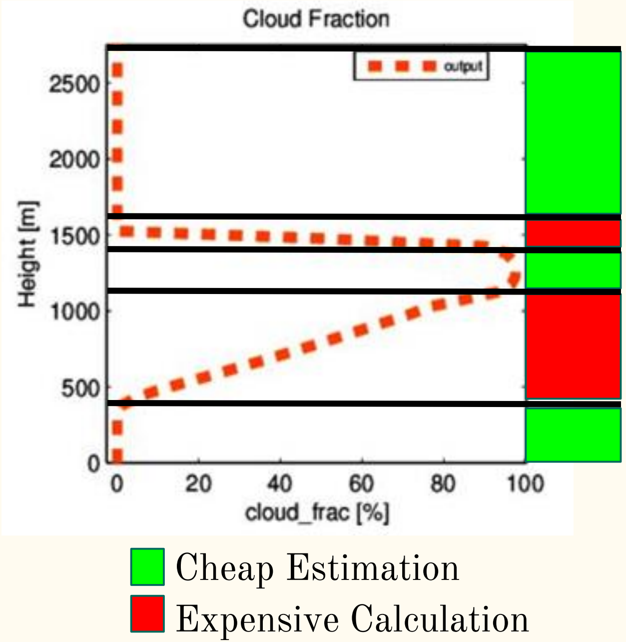
MKL_VML Functions Make the Cloud Fraction Calculation Much Faster

CLUBB computes a cloud faction based on the mean cloud water mixing ratio.

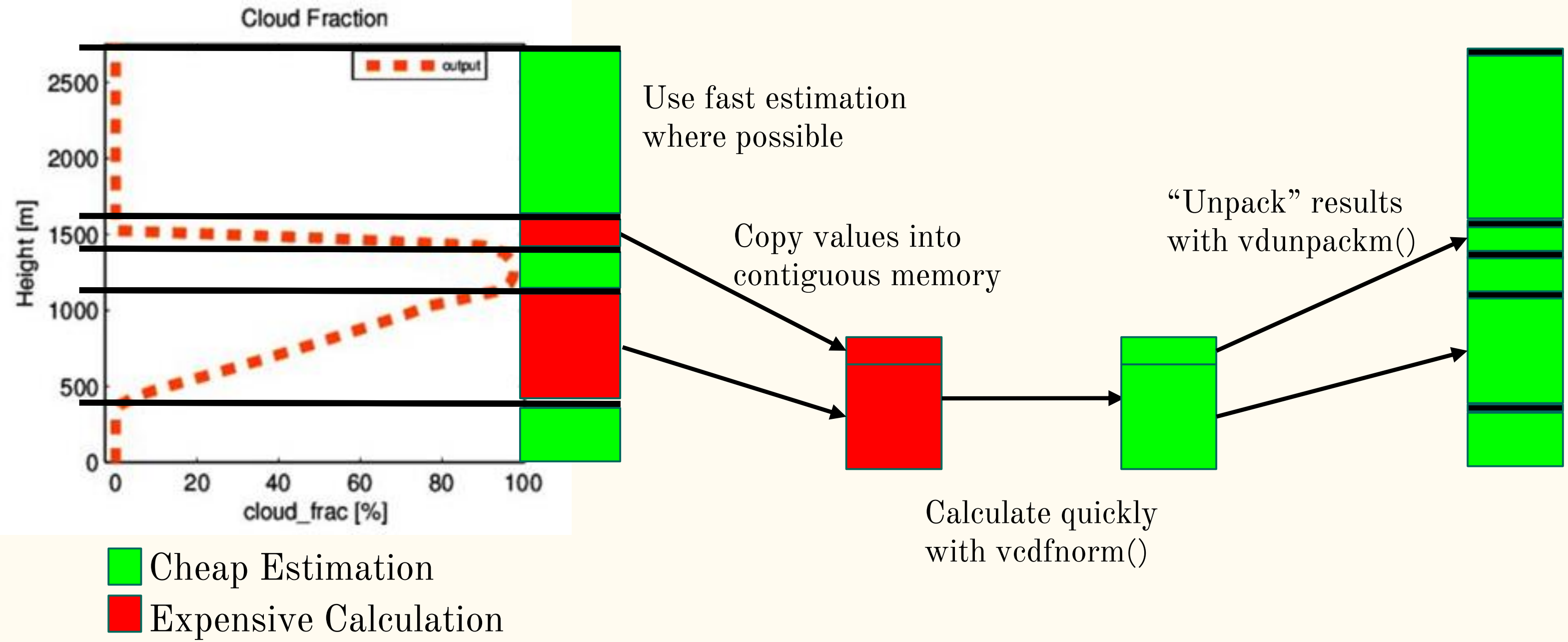
The cloud fraction is not significant on most grid levels.

Calculations using the expensive erf() function is only needed on a fraction of the levels.

Using vcdfnorm over all levels is less efficient than using the slow erf() on select levels.

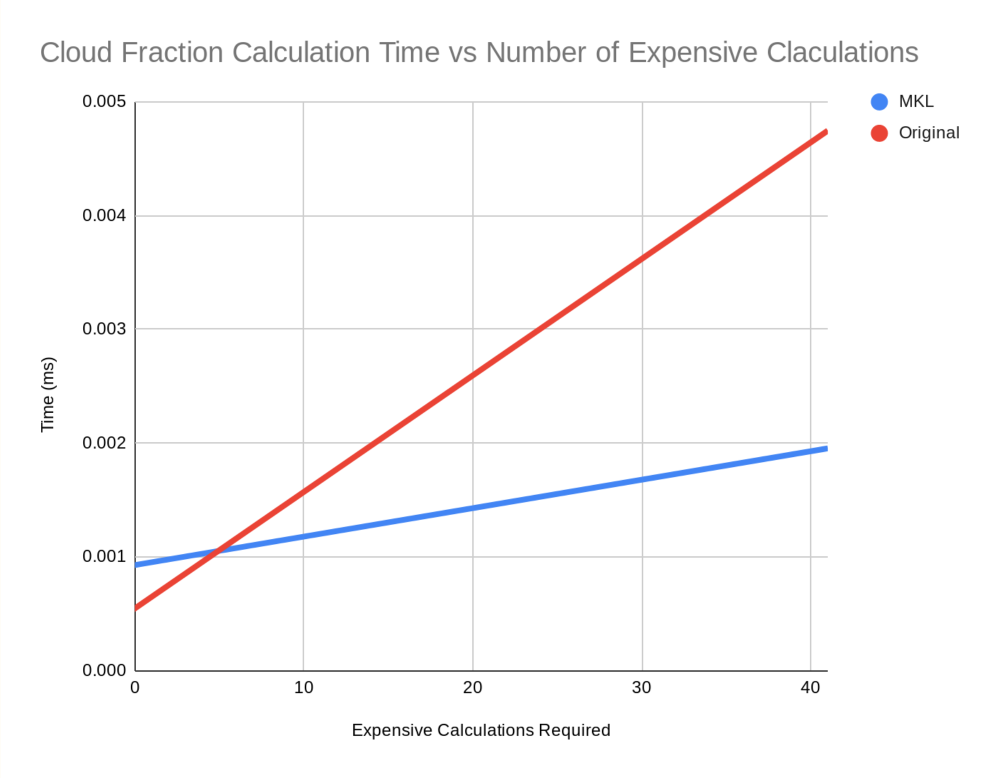


Cloud Fraction Calculation with MKL_VML Functions



The improvement in performance with this method depends on the number of grids levels requiring an expensive calculation, due to the extra packing step adding overhead.

MKL_VML Overhead Diminishes Quickly



The MKL_VML special function method performs better once more than 5 grid levels require an expensive calculation.

The number of number vertical levels requiring an expensive calculation is almost always great enough to make this refactoring improve computational efficiency.

The Mixing Length Calculation is not Vectorizable

CLUBB contains a calculation to estimate the mixing length between vertical levels.

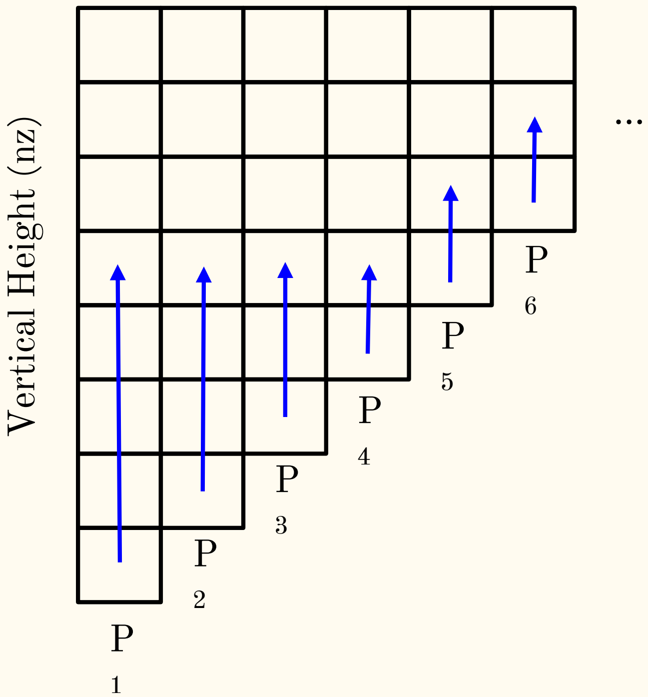
This is done by modeling a ‘parcel’ starting at each grid level, then determining how far that parcel may move by simulating the change in its turbulent kinetic energy (TKE).

The change in the TKE for a specific parcel at level n+1 depends on its change at level n.

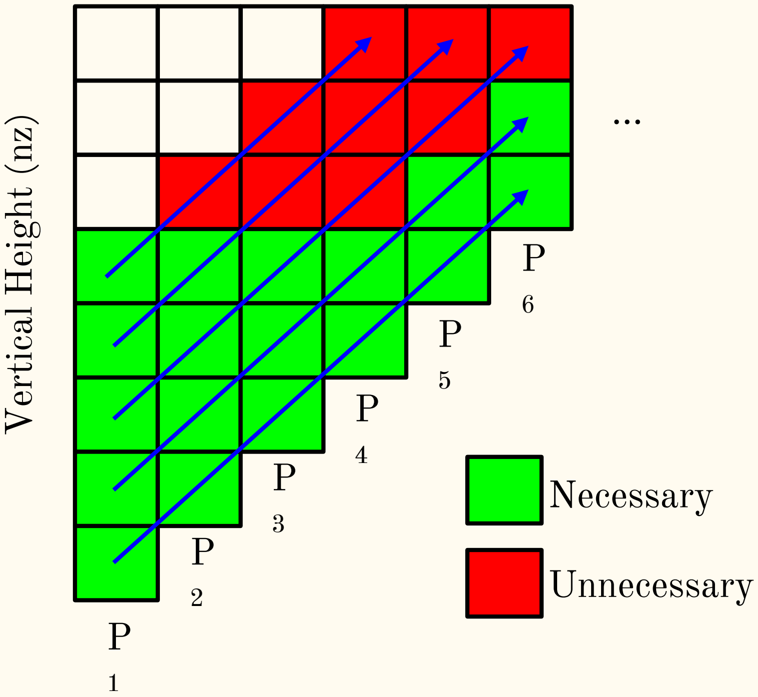
The calculation for a parcel ends once **TKE=0**.

Due to the uncertain stopping condition and data dependency, the calculation cannot be fully vectorized.

Visualization of the Mixing Length Calculation



Parcels starting at each nz are tracked up. These calculations have dependencies and can't vectorize.



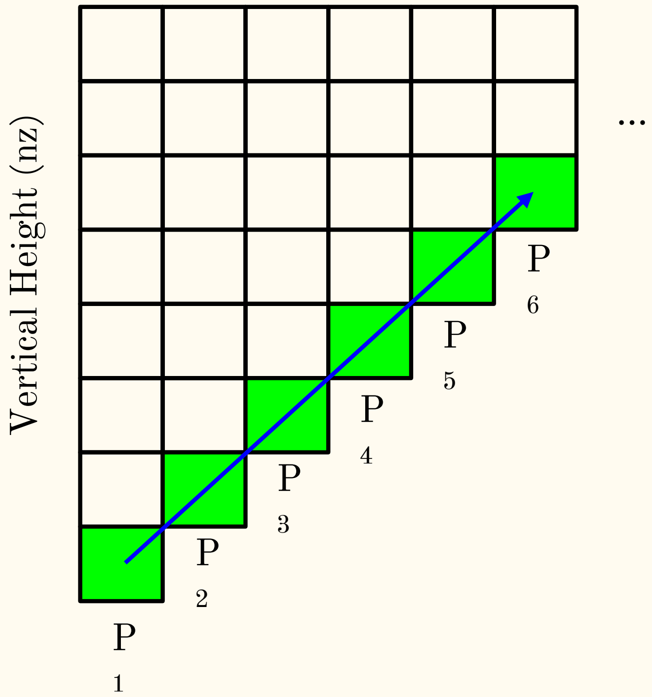
Vectorizing each calculation for each parcel is possible, but results in many extra calculations, ultimately degrading performance.

Non-vectorizable Calculations May be Partially Vectorizable

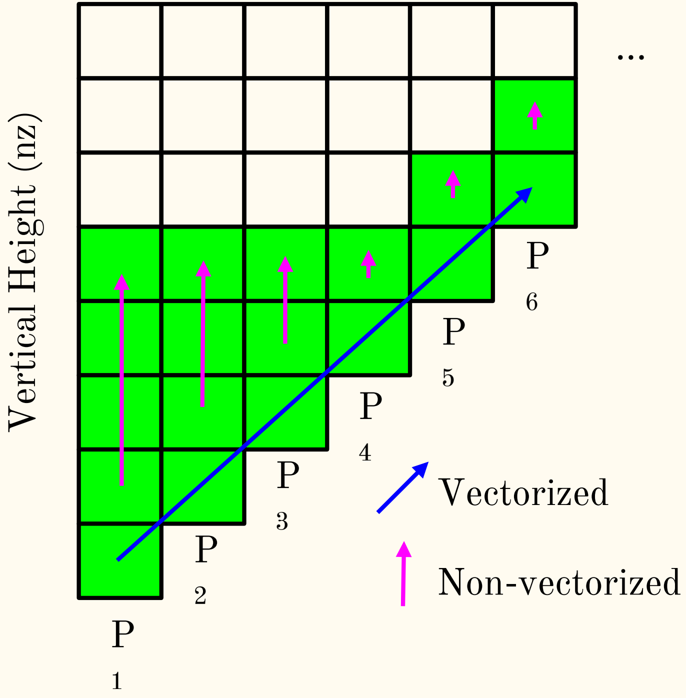
Fully vectorizing this calculation increases cost due to unnecessary calculations.

The first calculation of each parcel is **always** necessary.

Vectorizing the first calculations for each parcel reduces cost.



This Reduces the Cost of The Mixing Length Calculation in CLUBB by ~50%



This is works because not all parcels rise the same amount.

All calculations are necessary with this scheme.

There are less scalar instructions and more vectorized instructions.

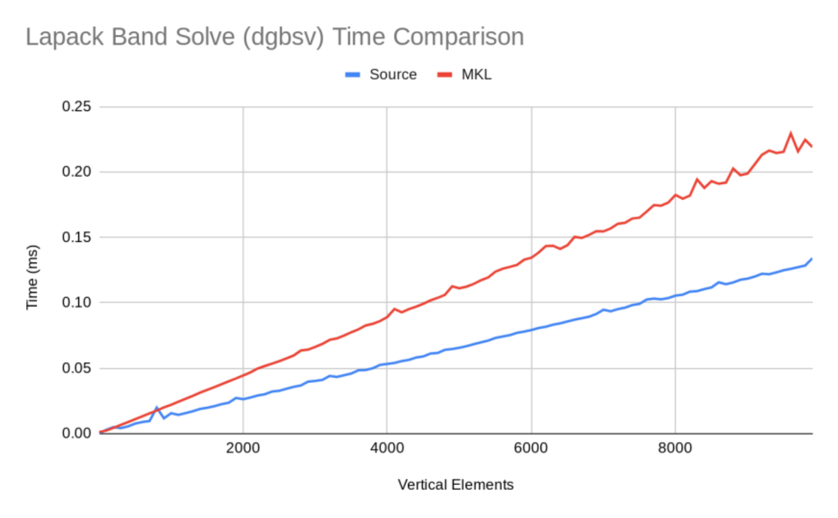
Lapack Source is More Efficient Than the MKL Library Implementation

CLUBB uses Lapack routines to solve large arrays.

The accepted approach is to use the well known Lapack methods.

There are two options; use Intel's MKL Lapack library or compile Lapack from source.

Source Lapack is faster on all systems, regardless of compiler options.



Small Changes Have Large Impacts

All the refactorings discussed here have been implemented in CLUBB.

Most microarchitectural optimizations do not produce bit-for-bit identical results, but are usually equivalent mathematically.

Over the past year, the cost of CLUBB is roughly 25% of what it used to be.