

Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«SIMD Vectorization»

Παναγιώτης Χατζηδούκας

SIMD (vector instructions)

- Recall SIMD: Single Instruction Multiple Data
 - we perform the same operation on many values at once.
- This was pioneered by the vector supercomputers
 - Cray X/MP
- Since 1999 part of all Intel CPUs
 - SSE (Streaming SIMD Extensions)
 - AVX (Advanced Vector Extensions)
- The easiest way of getting parallel speedup



SIMD registers and operations

- SIMD units contain vector registers
 - 128-bit registers XMM0 - XMM15 for SSE
 - 256-bit registers YMM0-YMM15 for AVX, overlapping the XMM registers

	255	128	0
YMM0		XMM0	
YMM1		XMM1	
YMM2		XMM2	
YMM3		XMM3	
YMM4		XMM4	
YMM5		XMM5	
YMM6		XMM6	
YMM7		XMM7	
YMM8		XMM8	
YMM9		XMM9	
YMM10		XMM10	
YMM11		XMM11	
YMM12		XMM12	
YMM13		XMM13	
YMM14		XMM14	
YMM15		XMM15	

- The SSE XMM registers can store

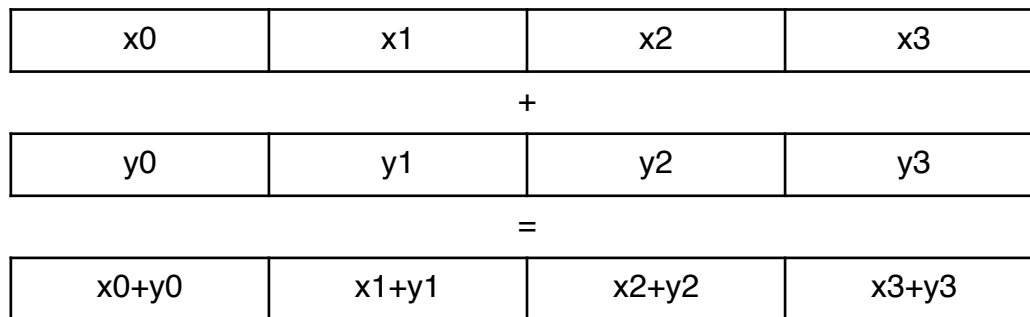
- XMM register
- 2 doubles
- 4 floats
- 2 64-bit integer
- 4 32-bit integers
- 8 16-bit integer
- 16 bytes

xmm															
x1								x2							
y1				y2				y3				y4			
i1								i2							
j1				j2				j3				j4			
k1		k2		k3		k4		k5		k6		k7		k8	

- AVX register can store 8 float or 4 double, integers since AVX2
- AVX-512 doubles that again

SIMD vector operations

- SIMD vector operations act on all values in the vector at once
- Example: adding four floats with one “packed floating point” instruction



- Advantages:
 - One instruction instead of 4
 - Memory access can be optimized
 - An easy way to gain speed for almost any code

SSE/AVX versions

- Intel and AMD have introduced more and more SIMD instructions with every new processor generation. The history is complex, and only roughly summarized below

Generation	Year	First Intel CPU	Main features
SSE	1999	Pentium III	
SSE2	2001	Pentium 4	SSE registers can be used together with scalar floating-point registers
SSE3	2004	Pentium 4 - Prescott	more instructions, and conversions between floating-point and integer
SSE4	2006	Core 2	More instructions
AVX	2011	Sandy Bridge	floating point 256 bit registers
AVX2	2013	Haswell	integer 256 bit registers
AVX-512	2017	Skylake	512 bit registers

SSE/AVX documentation

- The best documentation tool is by Intel, at
 - <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- An excellent documentation tool
 - lists all functions with clear explanation and documentation
 - allows to search for functions

The screenshot displays the Intel Intrinsics Guide interface. On the left, there are two sidebar menus: 'Technologies' and 'Categories'. The 'Technologies' menu lists various Intel architectures from MMX to Other, with SSE3 highlighted. The 'Categories' menu lists various function types from Application-Targeted to Trigonometry, with 'General Support' highlighted. The main content area shows a search result for the function `_mm_2intersect_epi32`. The search bar at the top contains the text `_mm_search`. The search results are displayed in a table with three columns: the function name, its signature, and its category. The function `_mm_2intersect_epi32` is listed with its signature `(__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)` and is categorized under `vp2intersectd`. Other functions listed include `_mm256_2intersect_epi32`, `_mm512_2intersect_epi32`, `_mm_2intersect_epi64`, `_mm256_2intersect_epi64`, `_mm512_2intersect_epi64`, `_mm512_mask_4dpwssd_epi32`, `_mm512_mask_4dpwssd_epi64`, `_mm512_mask_4dpwssds_epi32`, `_mm512_mask_4dpwssds_epi64`, `_mm512_mask_4fmaddd_ps`, `_mm512_mask_4fmaddd_ss`, `_mm512_mask_4fmaddd_q`, `_mm512_mask_4fmaddd_qd`, `_mm512_mask_4fmaddd_qd16`, `_mm512_mask_4fmaddd_qd32`, `_mm512_mask_4fmaddd_qd64`, `_mm512_mask_4fmaddd_qd128`, `_mm512_mask_4fmaddd_qd256`, `_mm512_mask_4fmaddd_qd512`, `_mm512_mask_4fmaddd_qd1024`, `_mm512_mask_4fmaddd_qd2048`, `_mm512_mask_4fmaddd_qd4096`, `_mm512_mask_4fmaddd_qd8192`, `_mm512_mask_4fmaddd_qd16384`, `_mm512_mask_4fmaddd_qd32768`, `_mm512_mask_4fmaddd_qd65536`, `_mm512_mask_4fmaddd_qd131072`, `_mm512_mask_4fmaddd_qd262144`, `_mm512_mask_4fmaddd_qd524288`, `_mm512_mask_4fmaddd_qd1048576`, `_mm512_mask_4fmaddd_qd2097152`, `_mm512_mask_4fmaddd_qd4194304`, `_mm512_mask_4fmaddd_qd8388608`, `_mm512_mask_4fmaddd_qd16777216`, `_mm512_mask_4fmaddd_qd33554432`, `_mm512_mask_4fmaddd_qd67108864`, `_mm512_mask_4fmaddd_qd134217728`, `_mm512_mask_4fmaddd_qd268435456`, `_mm512_mask_4fmaddd_qd536870912`, `_mm512_mask_4fmaddd_qd1073741824`, `_mm512_mask_4fmaddd_qd2147483648`, `_mm512_mask_4fmaddd_qd4294967296`, `_mm512_mask_4fmaddd_qd8589934592`, `_mm512_mask_4fmaddd_qd17179869184`, `_mm512_mask_4fmaddd_qd34359738368`, `_mm512_mask_4fmaddd_qd68719476736`, `_mm512_mask_4fmaddd_qd137438953472`, `_mm512_mask_4fmaddd_qd274877906944`, `_mm512_mask_4fmaddd_qd549755813888`, `_mm512_mask_4fmaddd_qd1099511627776`, `_mm512_mask_4fmaddd_qd2199023255552`, `_mm512_mask_4fmaddd_qd4398046511104`, `_mm512_mask_4fmaddd_qd8796093022208`, `_mm512_mask_4fmaddd_qd17592186044416`, `_mm512_mask_4fmaddd_qd35184372088832`, `_mm512_mask_4fmaddd_qd70368744177664`, `_mm512_mask_4fmaddd_qd140737488355328`, `_mm512_mask_4fmaddd_qd281474976710656`, `_mm512_mask_4fmaddd_qd562949953421312`, `_mm512_mask_4fmaddd_qd1125899906842624`, `_mm512_mask_4fmaddd_qd2251799813685248`, `_mm512_mask_4fmaddd_qd4503599627370496`, `_mm512_mask_4fmaddd_qd9007199254740992`, `_mm512_mask_4fmaddd_qd18014398509481984`, `_mm512_mask_4fmaddd_qd36028797018963968`, `_mm512_mask_4fmaddd_qd72057594037927936`, `_mm512_mask_4fmaddd_qd144115188075855872`, `_mm512_mask_4fmaddd_qd288230376151711744`, `_mm512_mask_4fmaddd_qd576460752303423488`, `_mm512_mask_4fmaddd_qd1152921504606846976`, `_mm512_mask_4fmaddd_qd2305843009213693952`, `_mm512_mask_4fmaddd_qd4611686018427387904`, `_mm512_mask_4fmaddd_qd9223372036854775808`, `_mm512_mask_4fmaddd_qd18446744073709551616`, `_mm512_mask_4fmaddd_qd36893488147419103232`, `_mm512_mask_4fmaddd_qd73786976294838206464`, `_mm512_mask_4fmaddd_qd147573952589676412928`, `_mm512_mask_4fmaddd_qd295147905179352825856`, `_mm512_mask_4fmaddd_qd590295810358705651712`, `_mm512_mask_4fmaddd_qd1180591620717411303424`, `_mm512_mask_4fmaddd_qd2361183241434822606848`, `_mm512_mask_4fmaddd_qd4722366482869645213696`, `_mm512_mask_4fmaddd_qd9444732965739290427392`, `_mm512_mask_4fmaddd_qd18889465931478580854784`, `_mm512_mask_4fmaddd_qd37778931862957161709568`, `_mm512_mask_4fmaddd_qd75557863725914323419136`, `_mm512_mask_4fmaddd_qd151115727451828646838272`, `_mm512_mask_4fmaddd_qd302231454903657293676544`, `_mm512_mask_4fmaddd_qd604462909807314587353088`, `_mm512_mask_4fmaddd_qd1208925819614629174706176`, `_mm512_mask_4fmaddd_qd2417851639229258349412352`, `_mm512_mask_4fmaddd_qd4835703278458516698824704`, `_mm512_mask_4fmaddd_qd9671406556917033397649408`, `_mm512_mask_4fmaddd_qd19342813113834066795298816`, `_mm512_mask_4fmaddd_qd38685626227668133590597632`, `_mm512_mask_4fmaddd_qd77371252455336267181195264`, `_mm512_mask_4fmaddd_qd154742504910672534362390528`, `_mm512_mask_4fmaddd_qd309485009821345068724781056`, `_mm512_mask_4fmaddd_qd618970019642690137449562112`, `_mm512_mask_4fmaddd_qd1237940039285380274899244224`, `_mm512_mask_4fmaddd_qd2475880078570760549798488448`, `_mm512_mask_4fmaddd_qd4951760157141521099596976896`, `_mm512_mask_4fmaddd_qd9903520314283042199193953792`, `_mm512_mask_4fmaddd_qd19807040628566084398387907584`, `_mm512_mask_4fmaddd_qd39614081257132168796775815168`, `_mm512_mask_4fmaddd_qd79228162514264337593551630336`, `_mm512_mask_4fmaddd_qd158456325028528675187103260672`, `_mm512_mask_4fmaddd_qd316912650057057350374206521344`, `_mm512_mask_4fmaddd_qd633825300114114700748413042688`, `_mm512_mask_4fmaddd_qd1267650600228229401496826085376`, `_mm512_mask_4fmaddd_qd2535301200456458802993652170752`, `_mm512_mask_4fmaddd_qd5070602400912917605987304341504`, `_mm512_mask_4fmaddd_qd10141204801825835211974608683008`, `_mm512_mask_4fmaddd_qd20282409603651670423949217366016`, `_mm512_mask_4fmaddd_qd40564819207303340847898434732032`, `_mm512_mask_4fmaddd_qd81129638414606681695796869464064`, `_mm512_mask_4fmaddd_qd162259276829213363391593739328128`, `_mm512_mask_4fmaddd_qd324518553658426726783187478656256`, `_mm512_mask_4fmaddd_qd649037107316853453566374957312512`, `_mm512_mask_4fmaddd_qd1298074214633706907132749914630224`, `_mm512_mask_4fmaddd_qd2596148429267413814265499829260448`, `_mm512_mask_4fmaddd_qd5192296858534827628530999658520896`, `_mm512_mask_4fmaddd_qd10384593717069655257061999317041792`, `_mm512_mask_4fmaddd_qd20769187434139310514123998634083584`, `_mm512_mask_4fmaddd_qd4153837486827862102824799726816768`, `_mm512_mask_4fmaddd_qd8307674973655724205649599453633536`, `_mm512_mask_4fmaddd_qd16615349947311448411299198907267072`, `_mm512_mask_4fmaddd_qd33230699894622896822598397814534144`, `_mm512_mask_4fmaddd_qd66461399789245793645196795629068288`, `_mm512_mask_4fmaddd_qd132922799578491587290393591258136576`, `_mm512_mask_4fmaddd_qd265845599156983174580787182516273152`, `_mm512_mask_4fmaddd_qd531691198313966349161574365032546304`, `_mm512_mask_4fmaddd_qd1063382396627932698323148730065092608`, `_mm512_mask_4fmaddd_qd2126764793255865396646297460130185216`, `_mm512_mask_4fmaddd_qd4253529586511730793292594920260370432`, `_mm512_mask_4fmaddd_qd8507059173023461586585189840520740864`, `_mm512_mask_4fmaddd_qd17014118346046923173170379681041481728`, `_mm512_mask_4fmaddd_qd34028236692093846346340759362082963456`, `_mm512_mask_4fmaddd_qd68056473384187692692681518724165926912`, `_mm512_mask_4fmaddd_qd136112946768375385385363037448321853824`, `_mm512_mask_4fmaddd_qd272225893536750770770726074896643707648`, `_mm512_mask_4fmaddd_qd544451787073501541541452149793287415296`, `_mm512_mask_4fmaddd_qd1088903574147003083082904299586574831392`, `_mm512_mask_4fmaddd_qd2177807148294006166165808599173149662784`, `_mm512_mask_4fmaddd_qd4355614296588012332331617198346299325568`, `_mm512_mask_4fmaddd_qd8711228593176024664663234396692598651136`, `_mm512_mask_4fmaddd_qd1742245718635204932932646879338519322272`, `_mm512_mask_4fmaddd_qd3484491437270409865865293758677038644544`, `_mm512_mask_4fmaddd_qd6968982874540819731730587517354077289088`, `_mm512_mask_4fmaddd_qd13937965749081639463461175034708154780176`, `_mm512_mask_4fmaddd_qd2787593149816327892692235006941630956352`, `_mm512_mask_4fmaddd_qd5575186299632655785384470013883261912704`, `_mm512_mask_4fmaddd_qd11150372599265311570768940027766523825408`, `_mm512_mask_4fmaddd_qd22300745198530623141537880055533047650816`, `_mm512_mask_4fmaddd_qd44601490397061246283075760111066095301632`, `_mm512_mask_4fmaddd_qd89202980794122492566151520222132190603264`, `_mm512_mask_4fmaddd_qd178405961588244985132303040444264381206528`, `_mm512_mask_4fmaddd_qd356811923176489970264606080888528762413056`, `_mm512_mask_4fmaddd_qd713623846352979940529212161777057524826112`, `_mm512_mask_4fmaddd_qd1427247692705959881058424323554115049652224`, `_mm512_mask_4fmaddd_qd2854495385411919762116848647108230099204448`, `_mm512_mask_4fmaddd_qd5708990770823839524233697294216460198408896`, `_mm512_mask_4fmaddd_qd11417981541647679048467394588432920396817792`, `_mm512_mask_4fmaddd_qd22835963083295358096934789176865840793635584`, `_mm512_mask_4fmaddd_qd45671926166590716193869578353731681587271168`, `_mm512_mask_4fmaddd_qd9134385233318143238773915670746336317454336`, `_mm512_mask_4fmaddd_qd18268770466636286477547831341492672634908672`, `_mm512_mask_4fmaddd_qd3653754093327257295509566268298534527981728`, `_mm512_mask_4fmaddd_qd7307508186654514591019132536597069055963456`, `_mm512_mask_4fmaddd_qd14615016373309029182038265073194138111926912`, `_mm512_mask_4fmaddd_qd29230032746618058364076530146388276223853824`, `_mm512_mask_4fmaddd_qd5846006549323611672815306029277655244770768`, `_mm512_mask_4fmaddd_qd11692013098647223345630612058555310489541536`, `_mm512_mask_4fmaddd_qd23384026197294446691261224117106210879082752`, `_mm512_mask_4fmaddd_qd4676805239458889338252244823421242175816504`, `_mm512_mask_4fmaddd_qd9353610478917778676504489646842484351633008`, `_mm512_mask_4fmaddd_qd18707220957835557353008979293684968703266016`, `_mm512_mask_4fmaddd_qd37414441915671114706017958587369937406532032`, `_mm512_mask_4fmaddd_qd74828883831342229412035917174739874813064064`, `_mm512_mask_4fmaddd_qd149657767662684458824071834349479749626128128`, `_mm512_mask_4fmaddd_qd299315535325368917648143668698959499252256256`, `_mm512_mask_4fmaddd_qd598631070650737835296287337397918998504512512`, `_mm512_mask_4fmaddd_qd1197262141301475670592574674795837997009025024`, `_mm512_mask_4fmaddd_qd2394524282602951341185149349591675994018050048`, `_mm512_mask_4fmaddd_qd4789048565205902682370298699183351988036100096`, `_mm512_mask_4fmaddd_qd9578097130411805364740597398366703976072200192`, `_mm512_mask_4fmaddd_qd19156194260823610729481194796733407952144400384`, `_mm512_mask_4fmaddd_qd38312388521647221458962389593466815904288800768`, `_mm512_mask_4fmaddd_qd76624777043294442917924779186933631808577601536`, `_mm512_mask_4fmaddd_qd153249554086588885835849558373867263617155203072`, `_mm512_mask_4fmaddd_qd306499108173177771671699116747734527234310406144`, `_mm512_mask_4fmaddd_qd612998216346355543343398233495469054468620812288`, `_mm512_mask_4fmaddd_qd1225996432692711086686796466990938089177241624576`, `_mm512_mask_4fmaddd_qd2451992865385422173373592933981876178354483249152`, `_mm512_mask_4fmaddd_qd490398573077084434674718586796375235670896498304`, `_mm512_mask_4fmaddd_qd980797146154168869349437173592750471341792996608`, `_mm512_mask_4fmaddd_qd1961594292288337738698874267185500942683585993216`, `_mm512_mask_4fmaddd_qd3923188584576675477397748534371001885367171986432`, `_mm512_mask_4fmaddd_qd7846377169153350954795497068742003770734343972864`, `_mm512_mask_4fmaddd_qd15692754338306701909590994137484007541468687945728`, `_mm512_mask_4fmaddd_qd31385508676613403819181988274968015082937375891456`, `_mm512_mask_4fmaddd_qd62771017353226807638363976549936030165874751782912`, `_mm512_mask_4fmaddd_qd125542034706453615276727953099872060321749503565824`, `_mm512_mask_4fmaddd_qd251084069412907230553455906199744120643499007131648`, `_mm512_mask_4fmaddd_qd502168138825814461106911812399488241286998014263296`, `_mm512_mask_4fmaddd_qd1004336277651628922213823647998962482579996028526592`, `_mm512_mask_4fmaddd_qd2008672555303257844427647295997924965159992057053184`, `_mm512_mask_4fmaddd_qd4017345110606515688855294591995849930319984114106368`, `_mm512_mask_4fmaddd_qd8034690221213031377710589183991699860639968228212736`, `_mm512_mask_4fmaddd_qd16069380442426062755421178367983399721279936456425504`, `_mm512_mask_4fmaddd_qd32138760884852125510842356735966799442559872912851008`, `_mm512_mask_4fmaddd_qd64277521769704251021684713471933598885119745825702016`, `_mm512_mask_4fmaddd_qd1285550435394085020433694269438671977702394916514044032`, `_mm512_mask_4fmaddd_qd2571100870788170040867388538877343555404789833028088064`, `_mm512_mask_4fmaddd_qd5142201741576340081734777077756870110809579666056176128`, `_mm512_mask_4fmaddd_qd10284403483152680163469554155513402221619159332112232256`, `_mm512_mask_4fmaddd_qd20568806966305360326939108311026804443238318664224464512`, `_mm512_mask_4fmaddd_qd411376139326107206538782166220536088864766373284488288`, `_mm512_mask_4fmaddd_qd822752278652214413077564332441072177729532746568977776`, `_mm512_mask_4fmaddd_qd1645504557304428826155128664822144355459065493137955552`, `_mm512_mask_4fmaddd_qd3291009114608857652310257329644288710918010986275911104`, `_mm512_mask_4fmaddd_qd6582018229217715304620514659288574218366021972551822208`, `_mm512_mask_4fmaddd_qd13164036458435430609241029318571484436732043945103644416`, `_mm512_mask_4fmaddd_qd26328072916870861218482058637142968873464087890207288832`, `_mm512_mask_4fmaddd_qd52656145833741722436964117274285937746928175780414577664`, `_mm512_mask_4fmaddd_qd1053122916674834448739282345485718754838563515608293328`, `_mm512_mask_4fmaddd_qd2106245833349668897478564690971437509677127031216586656`, `_mm512_mask_4fmaddd_qd4212491666699337794957129381942875019354254062433173312`, `_mm512_mask_4fmaddd_qd8424983333398675589914258763885750038708508124866346624`, `_mm512_mask_4fmaddd_qd16849966666797351179828517527771500077417162497332693248`, `_mm512_mask_4fmaddd_qd33699933333594702359657035055543000154834324994665386496`, `_mm512_mask_4fmaddd_qd67399866667189404719314070111086000309668649989330772992`, `_mm512_mask_4fmaddd_qd134799733334378809438628140222172000619337299978661545984`, `_mm512_mask_4fmaddd_qd269599466668757618877256280444344001238674599957323091968`, `_mm512_mask_4fmaddd_qd539198933337515237754512560888688002477349199914646183936`, `_mm512_mask_4fmaddd_qd107839786667530475550902511777776000495469839829292376784`, `_mm512_mask_4fmaddd_qd215679573335060951101805023555552000990939796585844753568`, `_mm512_mask_4fmaddd_qd431359146670121902203610047111104001981879593171689507136`, `_mm512_mask_4fmaddd_qd862718293340243804407220094222208003963759186343379014272`, `_mm512_mask_4fmaddd_qd1725436586680487608814440188444416007927518372686758028544`, `_mm512_mask_4fmaddd_qd3450873173360975217628880376888832015845136745373516057088`, `_mm512_mask_4fmaddd_qd6901746346721950435257760753777664031690273490747032114176`, `_mm512_mask_4fmaddd_qd13803492693443900870515521507555328`

Review: Caches

- Memory access speed did not keep up with Moore's law
- Are added to speed up memory access
 - Many GByte of slow but cheap DRAM
 - 2-20 MByte of fast L3-Cache
 - 256-512 kByte of faster L2-Cache per core
 - 2x32 - 2x64 kByte of fastest L1-Cache per core (instruction and data cache)
- Data that is read is stored in the caches and kept there until it needs to be evicted because new data is loaded
- Data written to memory is written to the cache and only further to memory if it needs to be evicted (or if we need to synchronize memory access between cores)
- Problems reusing memory will run faster!

Comparison of memory/cache speeds

- Data for Intel Sandy Bridge CPU

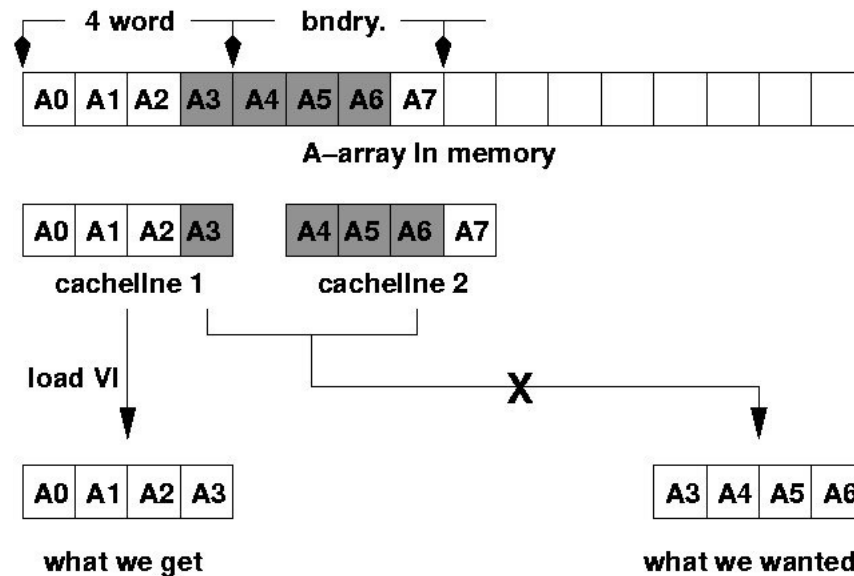
	Size	Access Time in cycles
L1 cache	2x32 KB	4-5
L2 cache	256 KB	12-19
L3 cache	3-20 MB	30-50
Memory	Many GB	~ 300

How does a cache work?

- CPU requests a word (e.g. 4 bytes) from memory
 - A full “cache line” (nowadays typically: 64 bytes) is read from memory and stored in the cache
 - The first word is sent to the CPU
- CPU requests another word from memory
 - Cache checks whether it has already read that part as part of the previous cache line
 - If yes, the word is sent quickly from cache to CPU
 - If not, a new cache line is read
- Once the cache is full, the oldest data is overwritten
- Locality of memory references are important for speed

Data alignment

- To achieve optimal speed data should be aligned on cache line boundaries.
- Consider what happens if we load one value that is not at the start of a cache line (on an old machine with 16 byte cache lines):



Alignment

- SSE registers are 16 bytes and need 16-byte alignment
- AVX registers are 32 bytes and need 32-byte alignment
- It is even better to align on cache line boundaries: 64/128 bytes on modern Intel CPUs

Allocating aligned data

- Aligned memory can be allocated
 - on POSIX (Linux, Unix) systems by calling `posix_memalign`
 - On Windows systems by calling `_aligned_malloc`
- Easiest using an alignment specifier in the declaration

```
float __attribute__((aligned(32))) sse[8];
```

Memory layout

- Array of Structures (AoS) vs Structure of Arrays (SoA)

```
#define N (128*1024*1024)
```

```
typedef struct {  
    double x;  
    double y;  
} point_t ;
```

```
point_t points[N];
```

```
for (int i = 0; i < N ; ++ i) {  
    points[i].x = drand48();  
    points[i].y = drand48();  
}
```

```
#define N (128*1024*1024)
```

```
struct {  
    double x[N];  
    double y[N];  
} points;
```

```
for (int i = 0; i < N ; ++ i) {  
    points.x[i] = drand48();  
    points.y[i] = drand48();  
}
```

- The points are stored in a array of structures (AoS)
- Typical object-based approach
- Data access is performed through objects

- The points are stored a structure of arrays (SoA)
- No object abstraction but focus on the data of the points
- Access to the individual information of the points

AoS vs SoA

```
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < N; ++i) {
    result += (points[i].x); // AoS
}
```

```
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < N ; ++i) {
    result += (points.x[i]); // SoA
}
```

- Ease of programming and performance
- AoS can cause unnecessary memory traffic
 - Pollutes the cache with unused data
- AoS not suitable for SIMD vectorization
 - Data transformation from AoS to SoA must be performed

When can a loop be vectorized?

- A loop can only be vectorized (or parallelized by threads) if there are no dependencies between the iterations:

- A linear congruential generator cannot be vectorized since one iteration depends on the previous one. We must wait for it to finish.

```
for (int i=1 ; i<N; ++i)
    rnd[i] = a* rnd[i-1] + c;
```

- adding vectors by saxpy can be vectorized (no dependencies)

```
for (int i=0 ; i<N; ++i)
    x[i] = a*x[i] + y[i];
```

- a lagged Fibonacci generator can be vectorized for vector lengths up to $\min(p,q)$. Dependencies only beyond a distance $\min(p,q)$

```
for (int i=std::max(p,q) ; i<N; ++i)
    rnd[i] = rnd[i-p] + rnd[i-q];
```

- Vector supercomputers had vector lengths up to 1024 elements.
- SSE has at most 16 bytes and AVX at most 32 bytes
 - the lagged Fibonacci is easier to vectorize

Detecting dependencies

- Look at every variable in the loop and check whether it might be written or read by another loop iteration. If so there is a dependency.
- Some dependencies can be removed by introducing additional variables

```
for (int i=0; i<N-1; i++) {  
    x = (b[i] + c[i])/2;  
    a[i] = a[i+1] + x;  
}
```

```
for (int i=0; i<N-1; i++)  
    a2[i] = a[i+1];
```

```
for (int i=0; i<N-1; i++) {  
    x = (b[i] + c[i])/2;  
    a[i] = a2[i] + x;  
}
```

- now both loops can be safely vectorized or parallelized
- Another special case are reductions:

```
double s=0;  
for (int i=0; i<N; i++)  
    s += x[i]*y[i];
```

- reductions can be vectorized, but it needs special care
- in OpenMP parallelization there is the reduction clause

Using SIMD instructions

- SIMD instructions can be used through assembly language. Complicated!
- Compilers offer support through intrinsics. Special types and functions that will be mapped directly to registers and SIMD instructions.
- Include the appropriate header or use the header `<x86intrin.h>` that is available with some compilers to load all headers available depending on the target platform

MMX	<code><mmintrin.h></code>
SSE	<code><xmmintrin.h></code>
SSE2	<code><emmintrin.h></code>
SSE3	<code><pmmintrin.h></code>
SSSE3	<code><tmmintrin.h></code>
SSE4.1	<code><smmintrin.h></code>
SSE4.2	<code><nmmintrin.h></code>
SSE4A	<code><ammintrin.h></code>
AES	<code><wmmintrin.h></code>
AVX	<code><immintrin.h></code>

- Enable code generation for SSE or AVX with the right compiler switches
- With gcc one option is to use the `-msse3`, `-msse4` or `-maxv` to enable SSE3, SSE4 or AVX support

Intrinsics: register data types

- The intrinsics headers define a few datatypes that map directly to SSE or AVX registers. The compiler will place such variables in the registers.
- Note: these start with two underscores!

<code>__m128</code>	4 floats
<code>__m128d</code>	2 doubles
<code>__m128i</code>	Integers of any size

<code>__m256</code>	8 floats
<code>__m256d</code>	4 doubles
<code>__m256i</code>	integers of any size, AVX2

<code>__m512</code>	16 floats
<code>__m512d</code>	8 doubles
<code>__m512i</code>	integers of any size, AVX-512

Intrinsics: naming of operations

- SSE and AVX instructions have a certain naming scheme
 - SSE operations: `_mm_name_type`
 - AVX operations: `_mm256_name_type`
 - operations on types shorter than a full register will not modify the higher bits

type	length in bits	description
ss	32	a single float
ps	128,256 or 512	4, 8, or 16 floats
sd	64	a single double
pd	128,256 or 512	2, 4, or 8 doubles
si64	64	any integers
si128	128	any integers
si256	256	any integers
pi8	64	8 8-bit integers
pi16	64	4 16-bit integers
pi32	64	2 32-bit integers
epi8	128,256 or 512	16, 32 or 64 8-bit integers
epi16	128,256 or 512	8, 16 or 32 16-bit integers
epi32	128,256 or 512	4, 8 or 16 32-bit integers
epi64	128,256 or 512	2, 4 or 8 64-bit integers

A first example: sscal

- Multiply a vector by a scalar, assuming aligned data and a vector length that is a multiple of 4

```
void sscal(int n, float a, float* x)
{
    // load the scale factor four times into a register
    __m128 x0 = _mm_set1_ps(a);
    // loop over chunks of 4 values
    for (int i=0; i<n/4; ++i) {
        __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
        __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
        _mm_store_ps(x+4*i,x2);         // store back aligned
    }
}
```

- We are using four instructions: two loads, a multiplication and a store

A first example: sscal

- Multiply a vector by a scalar, assuming aligned data, but now arbitrary vector length. We need to do the remaining values by hand

```
void sscal(int n, float a, float* x)
{
    // load the scale factor four times into a register
    __m128 x0 = _mm_set1_ps(a);

    int ndiv4 = n/4;
    // loop over chunks of 4 values
    for (int i=0; i<ndiv4; ++i) {
        __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
        __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
        _mm_store_ps(x+4*i,x2);         // store back aligned
    }

    // do the remaining entries
    for (int i=ndiv4*4 ; i< n ; ++i)
        x[i] *= a;
}
```

A first example: sscal

- Multiply a vector by a scalar, assuming aligned data, but now arbitrary vector length. We need to do the remaining values by hand

```
void sscal(int n, float a, float* x)
{
    // load the scale factor four times into a register
    __m128 x0 = _mm_set1_ps(a);

    int ndiv4 = n/4;
    // loop over chunks of 4 values
    for (int i=0; i<ndiv4; ++i) {
        __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
        __m128 x2 = _mm_mul_ps(x0,x1);  // multiply
        _mm_store_ps(x+4*i,x2);         // store back aligned
    }

    // do the remaining entries
    int i = ndiv4*4;
    switch (n-i) {
        case 3: x[i+2] *= a;
        case 2: x[i+1] *= a;
        case 1: x[i]  *= a;
    }
}
```

Load /store

- An incomplete summary of load/store instruction

Instruction	Types	Explanation
set1	all	sets all elements to a given value
set	all	set each element to a different value
setr	all	set in reverse order
setzero	pd, ps, si64,si128,si256	set to zero
load1	pd,ps	load a single value into each element of the register
broadcast	pd,ps	same as load1 but much faster (AVX only)
load	pd, ps, ss, sd, si128,si256	load values from memory into a register
loadr	pd,ps	load values in reverse order
loadu	pd, ps, ss, sd, si128,si256	load unaligned values from memory (slow!)
streamload	si128	load integer values bypassing the cache
store	pd, ps, ss, sd, si128,si256	store values from register into memory
storeu	pd, ps, ss, sd, si128,si256	store values from register into unaligned memory (slow!)
stream	pd, ps, pi,si128,si256	store values into memory bypassing the cache

- The streaming loads and stores bypass the cache. This reduces cache eviction, but it is hard to see a difference in many codes

Prefetch

- Prefetch instruction can be used to hint that some data will be used later and should already be fetched into the cache since they will soon be used

```
void _mm_prefetch (char const *p, int hint)
```

Hint	meaning
_MM_HINT_T0	prefetch into L1 (and L2 and L3) cache. Use for integer data.
_MM_HINT_T1	prefetch into L2 (an L3) cache. Use for floating point data.
_MM_HINT_T2	prefetch into L3 cache. Use if the cache line is not reused much.
_MM_HINT_NTA	prefetch into L2 but not L3 cache. Use if the data is needed only once.

- Example:

```
// loop over chunks of 4 values
for (int i=0; i<ndiv4; ++i) {
    _mm_prefetch((char*) y+4*i+8, _MM_HINT_NTA ); // prefetch data for two iterations later
    __m128 x1 = _mm_load_ps(x+4*i);              // aligned (fast) load
    __m128 x2 = _mm_mul_ps(x0,x1);                // multiply
    _mm_store_ps(x+4*i,x2);                       // store back aligned
}
```


Arithmetic floating-point instructions

- An incomplete summary of arithmetic instructions

Instruction	Explanation
add, sub	+, -
addsub	- on even, + on odd elements
mul, div	*, /
ceil	ceil, round up
floor	floor, round down
round	round, allows specification of rounding policy
min	min
max	max
rcp	reciprocal (inverse)
sqrt	sqrt
rsqrt	reciprocal (inverse) square root
and andnot	bitwise &, &!
or xor	bitwise , ^

Arithmetic integer instructions

- An incomplete summary of arithmetic instructions

Instruction	Explanation
add, adds	+, adds is saturated add: assigns maximum/minimum if overflow or underflow
sub, subs	-, subs is saturated sub: assigns maximum/minimum if overflow or underflow
avg	rounded average of x and y: $(x+y+1)/2$
mul	*, multiplies low words into result of twice the size - ignores every second input value
mullo	*, low word of product (result has twice the number of bits)
mulhi	*, high word of product (result has twice the number of bits)
sign	transfers sign of one integer to another and sets it to zero if “sign” is 0
min, max	min, max
and andnot	&, &!
or xor	, ^
sll, slli	<<, the version ending in i needs an integer constant shift
srl, slri	>> for unsigned integers, shifting in 0 bits

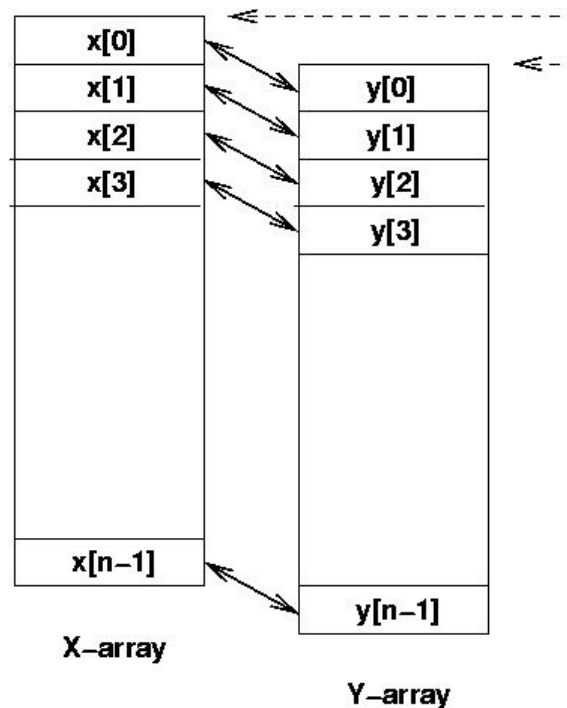
Comparisons

- An incomplete summary of important comparison instructions

Instruction	Types	Explanation
cmpeq, cmpneq	all	$x==y$, $x!=y$
cmpgt, cmpge	all	$x>y$, $x>=x$
cmplt, cmple	all	$x<y$, $x<=y$
cmpngt, cmpnge	floating point	$!(x>y)$, $!(x>=x)$
cmpnlt, cmpnle	floating point	$!(x<y)$, $!(x<=y)$
cmpord, cmpunord	floating point	tests whether the number are ordererd or unordered (e.g. if NaN)
test_all_ones	i128	test if all bits are 1
test_all_zeros	i128	test if all bits are 0
test_mix_ones_zeros	i128	test if either all are 0 or all are 1

_axpy operations

- Alignment is trickier with operations involving two vectors
 - Example `_axpy`:
$$\vec{y} = a \vec{x} + \vec{y}$$
 - We need both arrays aligned in the same way
 - Two solutions:
 - either always require alignment
 - or code a slow version to use if not aligned



saxpy

- a vectorized saxpy implementation assuming alignment

```
void saxpy(int n, float a, float* x, float* y)
{
    // load the scale factor four times into a register
    __m128 x0 = _mm_set1_ps(a);

    int ndiv4 = n/4;
    // loop over chunks of 4 values
    for (int i=0; i<ndiv4; ++i) {
        __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
        __m128 x2 = _mm_load_ps(y+4*i); // aligned (fast) load
        __m128 x3 = _mm_mul_ps(x0,x1); // multiply
        __m128 x4 = _mm_add_ps(x2,x3); // add
        _mm_store_ps(y+4*i,x4); // store back aligned
    }

    // do the remaining entries
    for (int i=ndiv4*4; i< n; ++i)
        y[i] += a*x[i];
}
```

sdot

- a vectorized dot product assuming alignment
- we have to manually do the reduction

```
float sdot(int n, float* x, float* y)
{
    // set the total sum to 0, one sum per vector element
    __m128 x0 = _mm_set1_ps(0.);

    // loop over chunks of 4 values
    int ndiv4 = n/4;
    for (int i=0; i<ndiv4; ++i) {
        __m128 x1 = _mm_load_ps(x+4*i); // aligned (fast) load
        __m128 x2 = _mm_load_ps(y+4*i); // aligned (fast) load
        __m128 x3 = _mm_mul_ps(x1,x2);  // multiply
        x0 = _mm_add_ps(x0,x3);          // add
    }

    // store the 4 partial sums back to aligned memory
    float __attribute__((aligned(32))) tmp[4];
    _mm_store_ps(tmp,x0);

    // do the reduction over the vector elements by hand
    float sum = tmp[0]+tmp[1]+tmp[2]+tmp[3];

    // do the remaining entries
    for (int i=ndiv4*4 ; i< n ; ++i)
        sum += x[i]*y[i];

    return sum;
}
```

Matrix-vector multiplication

- Given that x, y are appropriately aligned and $n \% 8 == 0$

```
#include <string.h>
#include <immintrin.h>

void mxv(int m, int n, float * restrict A, float * restrict x,
        float * restrict y)
{
    __m256 AA, xx, yy;
    int simd_width = 8;

    memset(y, 0, m * sizeof(float));
    for (int i=0; i<m; ++i) {
        yy = _mm256_set_ps(0, 0, 0, 0, 0, 0, 0, 0);

        for (int j=0; j<n; j+=simd_width) {
            AA = _mm256_load_ps(A+i*n+j);
            xx = _mm256_load_ps(x+j);
            yy = _mm256_fmadd_ps(AA, xx, yy); // yy = AA*xx + yy
        }

        y[i] = yy[0]+yy[1]+yy[2]+yy[3]+yy[4]+yy[5]+yy[6]+yy[7];
    }
}
```

Automatic vectorization with gcc

- Modern compilers try to automatically vectorize loops. This can save you time but will sometimes not be as good as vectorization by hand.
- Compiler options for gcc/g++
 - Turn vectorization on: `-ftree-vectorize`
 - Generate vectorization reports: `-Om -ftree-vectorizer-verbose=n`

n	Description
0	No output at all.
1	Report vectorized loops.
2	Also report unvectorized "well-formed" loops and respective reason.
3	Also report alignment information (for "well-formed" loops).
4	Like level 3 + report for non-well-formed inner-loops.
5	Like level 3 + report for all loops.
6	Print all vectorizer dump information.

- Further reading
 - GNU documentation
 - <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>
 - Critical analysis of what autovectorization in gcc can and cannot do:
 - <http://locklessinc.com/articles/vectorize/>

Aliasing prevents optimization

- Consider the saxpy operation:

```
void saxpy(int n, float a, float* x, float* y)
{
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

- Naïvely it seems this can be vectorized since there are no dependencies: each iteration accesses different elements
- Now consider the following call:

```
float x[1000];
saxpy(999, 1., x, x+1)
```

- Problem: now $y=x+1$ and we have an “aliasing” problem. The loop becomes

```
for (int i=0; i<n; ++i)
    x[i+1] += a*x[i];
```

- We have potential dependencies! No optimization or vectorization is actually possible unless we prevent aliasing.

restrict

- Fortran-77 can optimize aggressively since aliasing is forbidden
- Fortran-90 and later, C, C++, ... have pointers and with pointers aliasing becomes a potential problem and prevents many optimizations.
- Solution in C: restrict keyword to declare that pointers are not aliased.

```
void saxpy(int n, float a, float* restrict x, float* restrict y)
{
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

- The compiler now assumes no aliasing.
- Note that the compiler does not check for aliasing. The caller must be careful!

Declaring alignment

- In our manually vectorized code we assumed the absence or presence of alignment. We can also tell this to the compiler:
- gcc/g++ have the following extension

```
__builtin_assume_aligned(variable,alignment);
```

```
void saxpy(int n, float a, float* restrict x, float* restrict y)
{
    __builtin_assume_aligned(x,32);
    __builtin_assume_aligned(y,32);
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

Vectorization with OpenMP

- The `simd` construct (OpenMP 4.0+)
 - `#pragma omp simd`: declares that a loop will be utilizing SIMD

```
float a[8], b[8];
...
#pragma omp simd
for (int n=0; n<8; ++n) a[n] += b[n];
```
 - `#pragma omp declare simd`: indicates a function or procedure that is explicitly designed to take advantage of SIMD parallelism.
 - The compiler may create multiple versions of the same function for different CPU capabilities for SIMD processing
 - `aligned` attribute: hints the compiler that each element listed is aligned to the given number of bytes.

```
#pragma omp declare simd aligned(a,b:16)
void add_arrays(float *__restrict__ a, float *__restrict__ b)
{
    #pragma omp simd aligned(a,b:16)
    for (int n=0; n<8; ++n) a[n] += b[n];
}
```

- The compiler attempts to vectorize regardless of the OpenMP `simd` directives

Vectorization with OpenMP

- collapse clause: can be added to bind the vectorization into multiple nested loops

```
#pragma omp simd collapse(2)
for (int i=0; i<4; ++i)
    for (int j=0; j<4; ++j)
        a[j*4+i] += b[i*4+j];
```

- reduction clause: can be used with SIMD just like with parallel loops

```
int sum=0;
#pragma omp simd reduction(+:sum)
for (int n=0; n<1000; ++n) sum += table[n];
```

Vectorization with OpenMP

- The for simd construct: for and simd can be combined
 - divide the execution of a loop into multiple threads
 - execute the loop slices in parallel using SIMD

```
float sum(float* table)
{
    float result=0;
    #pragma omp parallel for simd reduction(+:result)
    for (int n=0; n<1000; ++n)
        result += table[n];

    return result;
}
```

References

- Prof. M. Troyer, HPCSE I, ETH Zurich
- https://en.wikipedia.org/wiki/Fast_inverse_square_root
- https://www.mathworks.com/help/simulink/ref_extras/hdlreciprocal.html